1984 UniForum Conference Proceedings

/usr/group
USENIX Association

/usr/group
The International Network of UNIX Users

UniForum Conference Proceedings

Washington D.C.

January 17-20. 1984

1984

/usr/group

USENIX Association

U N I X

January 1984
UniForum Conference Proceedings
Washington, DC

/usr/group
USENIX Association

Preface

The January 1984 UniForum meeting was held from Tuesday, January 17 to Friday, January 20, 1984, in Washington, DC. UniForum was a joint meeting of the /usr/group and USENIX Association to share information about UNIX and UNIX-like Operating Systems.

While the following pages list all presentations made at UniForum, these *Proceedings* contain the complete text of only those papers submitted for publication (as indicated by a page reference).

The papers are arranged in order, by session, as presented. An author index and keyword index of titles is provided in the back of the book.

For more information on either sponsoring group, write to the addresses below. For information regarding additional copies of these *Proceedings*, contact the USENIX Association.

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710
510/528-8649

UniForum
2901 Tasman Drive, #210
Santa Clara, CA 95054
408/986-8840

UniForum Conference Schedule

Tuesday, January 17, 1984

Session 1
Software Contracts & Licenses
Susan H. Nycum – Gaston, Snow, Ely Bartlett

Session 2
C Style and Portability
Eric Allman – Britton-Lee

Session 3
UNIX Systems Administration
Ed Gould, Bob Kridle – mt Xinu

Session 4
Advanced Shell Programming
Steve Bourne – Silicon Graphics

Session 5
Vi Editor
Auxco

Session 6
UNIX Systems on Local Area Networks

Wednesday, January 18, 1984

Session 7
Keynote Address — Growth of the UNIX Market
Jack Scanlon – AT&T Technologies

Session 8
Joint Session

Session 9
UNIX in Government
Chair: Josephus P. Knippenberg – FCC

UNIX-Based Database Management Systems
Rick Kanner – FCC
John Brinkema – Federal Judiciary Center

Driver-Based Protocol Implementations
Greg Chesson – Silicon Graphics

Session 10
Networks — Aspects of Networking Under UNIX
Chair: Thomas Ferrin – University of California, San Francisco

The Excelan TCP/IP Protocol Package 1
Bill Northlich – Northlich CSS
Bruce Borden – Silicon Graphics

Worknet: A XENIX-Based Merged Filesystem Network
Alan Greenspan – Altos Computer Systems

CSNET Grows Up 13
Michael T. O'Brien – The Rand Corporation
Daniel B. Long – Bolt Beranek and Newman

Session 11
The Future of UNIX
Chair: Jean Yates– Yates Ventures

Panel Discussion:
Industry Analysts Present Marketing Projections and Key Trends
Dataquest, Gnostic Concepts, Yankee Group, Yates Ventures

Session 12
Distributed UNIX — Multiprocessing Under UNIX
Chair: Alan Nemeth – PrimeComputer

**Software Administration over Computer Networks —
The Exptools Experience** 17
Joseph L. Steffen – AT&T Bell Laboratories

A Distributed File System for UNIX 23
*Matthew S. Hecht, John R.Levine, Justin C. Walker –
Interactive Systems Corporation*

Multiprocessor Debugging on a Shared Memory System ... 33
Chet Britten, Paul Chen – Metheus Corporation

Panel Discussion:
Multiprocessing Issues

Thursday, January 19, 1984

Session 13
Microcomputer to Mainframe Communications
Chair: David Buck – D.L. Buck & Associates

Adapting UNIX for Data Communications 39
Charles M. Robins – Rabbit Software Corporation

Connecting a UNIX System to an X.25 Network 47
Joaquin Miller – Pacific Software Manufacturing Company

**Bisynch Batch & Interactive Communications —
Popular Ways to Connect Micros to Other Computers**
David Buck – D.L. Buck & Associates

Session 14
**Compilers and Languages —
New Languages and News About Familiar Languages**
Chair: Louis Salkind – New York University

The Evolution of the Portable C Compiler
S.C. Johnson – AT&T Bell Laboratories

You CAN Feel Good Knowing It Is Written In C 57
Alan R. Feuer – Catalytix Corporation

An Implementation of the B Programming Language 65
Lambert Meertens, Steven Pemberton – CWI

Object-Oriented Programming in C Language
Brad J. Cox – Productivity Products

Session 15
Graphics Applications on Workstations
Chair: Larry Tarbell – National Security Agency

**Interactive Mapped Displays in the Apollo DOMAIN
Environment**
James Wilson – National Security Agency

**Powerful Workstations as a Tool for Teaching
Computer Science**
Eric Wolf – Brown University

A Network Window-Manager
James Gosling, David S.H. Rosenthal – Carnegie/Mellon University

VLSI Layout in a UNIX Environment
Dr. Bruce Brown – CAECO, a division of Chromatics

Session 16
**Over Under Sideways Down,
Backwards Forwards Square & Round.
UNIX Directions**
Chair: Brian Redman – AT&T Bell Laboratories

Behind Every Binary License Is the UNIX Heritage 75
*Brian E. Redman – Central Services Organization
Pat E. Parseghian – Princeton University*

Political History of UNIX 83
Andrew Tannenbaum – MASSCOMP

Proposed Syntax Standard for UNIX System Commands ... 89
Kathleen Hemenway, Helene Armitage – AT&T Bell Laboratories

Decision Avoidance: UNIX from DEC, Finally
Armando Stettner – Digital Equipment Corporation

Session 17
UNIX to Microports
Chair: Michael P. Florio – Palantir

Panel Discussion
Update on Progress of Microports
*Tom Beaver – Motorola
Robert Freund – National Semiconductor
Bill Maxey – Intel
David Guzeman – Zilog
Tom Crowley – Western Electric*

Session 18
Applications
Chair: Reidar Bornholdt – Columbia University

MLE (Multi-Lingual Editor)
Scott Bradner – Harvard University

**MPS: A UNIX-Based Microcomputer Message
Switching System** 99
T. Scott Pyne, Joseph S.D. Yao – Hadron

Taming the Beast (An RSX Emulator for UNIX)
Daniel R. Strick – University of Pittsburgh

A UNIX-Based Color Graphics Workstation 115
Rex McDowell – Metheus Corporation

User-Level Window Managers for UNIX 123
Robert J.K. Jacob – Naval Research Laboratory

Session 19
New Developments in Office Automation
Chair: Amy Wohl – Advanced Office Concepts Corporation

The E2 Editor System
Ned Irons – Interactive Systems

**Workstation Systems in the Corporation —
Office Automation for 1984**
Bill Stensend – Sydis

Session 20
Implementations
Chair: Michael O'Dell – Group L

The UNIX Paging System
Keith Kelleman – AT&T Bell Laboratories

Providing a Job Control Facility in UNIX System V
W.P. Weber Jr., L.S. Weisbrot – AT&T Bell Laboratories

Reloadable UNIX Device Drivers 135
Thomas Alborough – Creare R&D

OSx: Towards a Single UNIX System for Superminis 145
Ross Bott – Pyramid Technology

**New 1/2-Inch Tape Options and Trade-Offs for
4.2BSD UNIX on DEC VAX Processors** 168
Bob Kridle – mt Xinu

Friday, January 20, 1984

Session 21
Case Studies in Office Automation
Chair: Amy Wohl – Advanced Office Concepts Corporation

Speakers:
William Kaynor – Davis, Polk and Wardwell
Ron Jones – BEFCOM
David Del Monte – World Bank

Session 22
Implementations
Chair: Joseph S.D. Yao – Hadron

A Layered Implementation of the UNIX Kernel on the HP9000 Series 500 Computer 183
Jeff Lindberg– Hewlett-Packard

Porting XENIX to the Unmapped 8086
John Bruno Hare, Dean Thomas – The Santa Cruz Operation

Writing Device Drivers for XENIX Systems 195
Jean McNamara, Paresh Vaish, Richard N. Bryant – Intel Corporation

Transparent Implementation of Shared Libraries 209
Curtis B. Downing – Bunker Ramo Information Systems
Frank Farance – Systems Theory Design Corporation

UNIX Block I/O Optimization on Microcomputers 223
David Robboy – Intel Corporation

Session 23
UNIX Standards
Chair: Heinz Lycklama – Interactive Systems Corporation

/usr/group Standards Effort 231
Heinz Lycklama – Interactive Systems Corporation

File/Record Locking Primitive
John Bass – Consultant

C Language Issues
Bill Plauger – Whitesmiths

Validation Suites
Richard Hammons – Pyramid Technology

Extensions to the Standard
David Buck – D.L. Buck & Associates

Session 24
Communications — UNIX Talks to Itself and Others
Chair: Mark Horton – AT&T Bell Laboratories

A UNIX to VAX/VMS Communications Link
Clifford N. Cary – Creare R&D

Loving UUCP, or How I Spent My Summer Vacation
P. Honeyman, D.A. Nowitz, B.E. Redman – AT&T Bell Laboratories

Experimental Implementation of UUCP — Security Aspects 245
D.A. Nowitz, P. Honeyman, B.E. Redman – AT&T Bell Laboratories

Mapping the UUCP Network 251
Rob Kolstad, Karen Summers-Horton – AT&T Bell Laboratories

UUCP-USENET Panel:
UUCP-USENET Issues

Session 25
UNIX on PCs — Good for What?
Chair: Bruce Weiner – Consultant

Speakers:
Jeff Schreibman – Unisoft
Bob Swartz – Mark Williams Company
Dick Churchill – Unisource

Session 26
Graphics Under UNIX
Chair: Noel Kropf – Columbia University

UNIX as a Development Base for High-Performance Graphics Applications
Thomas Ferrin – University of California, San Francisco

DAPS and GSDL: A Procedural Approach to Graphics
Christos Tountas – Graphics Information Systems Technology

The Design of a Dedicated Graphics Subsystem for a UNIX Machine
Jack Burness – MASSCOMP

Image Processing Work Station
Dale Hensley – TRW

Session 27
Applications Generators Under UNIX
Chair: Ellen Ullman – Insurnet

Building Tunnels and Bridges: Constructing a Commercial Application Under UNIX 259
Ellen Ullman, Page Thompson, Jerry Carlin – Insurnet

Applications Generators Accelerate UNIX Acceptance Through Rapid Applications Software Development
Steven M. Thomas – The Software Express

SofForms
Sol Goldenberg – SofTest

Robot Tools for Automating Information Assembly Processes
Tony Olekshy – Olekshy Hoover & Associates

Session 28
Realtime and Other Kernel Mods
Chair: Reidar Bornholdt – Columbia University

UNIX Performance Optimization — UNIX/68000/SKYFFP
Joseph F. Germann – Sky Computers

Life with UNIX in Real-Time 285
Steven T. Polyak, Jeffrey S. Barr – Contel Information Systems

Real-Time Extensions to the UNIX Operating System 293
Byron Look, Gary Ho – Hewlett-Packard

Integrating a Peripheral Floating Point Processor in UNIX
Eryk Vershen – Unisoft

Vendor Exhibits

COMPANY NAME: AT&T Technologies
ADDRESS: P.O. Box 25000
Greensboro, NC 27420
TELEPHONE NUMBER: 1-800-828-UNIX
MARKETING CONTACT: O.L. Wilson
Manager, Software
Sales and Marketing

Products exhibited:

- *UNIX System V (Release 2.0)* — enhancements include a 5 to 10% improvement in overall performance, more than 500 fixes, new and enhanced commands, new utilities such as job control, and improved documentation.
- *UNIX Documenter's Workbench Software* — complete text processing with device independent troff.
- *UNIX Writer's Workbench Software* — more than 25 powerful programs for all types of writing.
- *UNIX Instructional Workbench Software* — UNIX System V courseware delivery system.
- *UNIX System V* — support and training.

New products announced:
UNIX System V (Release 2.0)
UNIX Documenter's Workbench Software
BASIC Interpreter
MC 68000 Software Generation Systems

Other announcements:

AT&T Technologies and Digital Research Inc. announce joint development of an applications library for UNIX System V. Library wilg evaluate and market UNIX System V applications from independent software vendors (ISVs).

Questions to:
W.E. Swain
1-800-828-UNIX

COMPANY NAME: AVIV Corporation
ADDRESS: 26 Cummings Park
Woburn, MA 01801
TELEPHONE NUMBER: (617) 933-1165
MARKETING CONTACT: Ed Arsenault

AVIV Corporation designs, manufactures and markets high-performance controllers and subsystems for DEC, DGC and Multibus-based computers. Tape controllers are compatible with all leading models of industry-standard streaming and start/stop drives. In addition, AVIV has the broadest range of GCR (6250 bpi) compatible peripherals, including CDC, STC and Telex drives. AVIV disk controllers are available with standard SMD drives from 80-675 MB, and with fixed or removable media. The UNIBUS and VAX 11/750 CMI bus controllers also support advanced SMD drives, such as the CDC 9715-500 and the Fujitsu Eagle, which have a 1.8 MB/second transfer rate. AVIV controllers are compatible with all DEC and DGC operating systems, as well as UNIX and UNIX-derivative operating systems.

COMPANY NAME: Able Computer
ADDRESS: 1732 Reynolds
Irvine, CA 92714
TELEPHONE NUMBER: (714) 979-7030
MARKETING CONTACT: Bob Nicholson

Communication solutions for DEC and IBM systems. Before installing networks, switches, multiplexers or other communication systems or devices Able may provide a more cost-effective and longer range solution. Able's selection of software-transparent solutions are the most advanced and cost-effective in the industry. If your problem is connectivity, see Able.

COMPANY NAME: Absolut Software
ADDRESS: 2001 Beacon Street
Boston, MA 02146
TELEPHONE NUMBER: (800)-ABS-UNIX,
(617) 277-0610
MARKETING CONTACT: John T. McGrath

Absolut offers a line of complete inventory management and business systems for chainstore retailers, mail-order houses, distributor/wholesalers, manufacturers. Software is written in "C", incorporates the Unify data base and Report Generator and will run on any 68000 or larger computer.

COMPANY NAME: Amdahl Corporation
ADDRESS: P.O. Box 3470
Sunnyvale, CA 94088-3470
TELEPHONE NUMBER: (408) 746-8945
MARKETING CONTACT: Cynthia Ainsworth

The Amdahl Universal Timesharing System (UTS) combines the functions of UNIX with the power of a large mainframe. UTS runs on any S/370 architecture processor, including both Amdahl and IBM CPUs.

COMPANY NAME: Apollo Computer, Inc.
ADDRESS: 15 Elizabeth Drive
Chelmsford, MA 01824
TELEPHONE NUMBER: (617) 256-6600
MARKETING CONTACT: John Bowne Jr.

Apollo Computer demonstrated AUX, the firm's adaptation of UNIX System III software with Berkeley extensions. Operating in a multi-mode local-area network, AUX allows users full Bell and Berkeley UNIX functionality in addition to Apollo's standard windowing, networking, and graphics capabilities.

The actual hardware consisted of two DN300 monochromatic and one DN600 color, computational nodes. One of the DN300s was running "diskless", thus showing Apollo's unique capability of supporting virtual memory, via demand paging, over a local area, 12-megabit-per-second, network.

COMPANY NAME: Applied Data Systems, Inc.
ADDRESS: 9811 Mallard Drive, Suite 213
Laurel, MD 20708
TELEPHONE NUMBER: (301) 953-9326
MARKETING CONTACT: Robert Kondner

Applied Data Systems, Inc., exhibited the VectorScan 512 color graphic controller that sells for $975.00 at quantity one. This unit provides 512 x 480 pixel resolution at 4 bits per pixel. Both monochrome and color monitors are supported by the VectorScan 512. Also, hard copies of the graphic images can be obtained on various low-cost dot matrix printers.

COMPANY NAME: Avalon Computer Systems, Inc.
ADDRESS: 55 North St. John Avenue
Pasadena, CA 91103
TELEPHONE NUMBER: (818) 796-0861
MARKETING CONTACT: Conrad Schneiker

Avalon demonstrated its VAX-attached processor system. This system is available now as an addition to VAX systems. Adding one or more Avalon systems dramatically increases the computing power of a timesharing system. Each Avalon SP/10 processor includes one million bytes of memory and also includes floating point hardware. The SP/10 is completely contained on one UNIBUS circuit board and provides computing power similar to the 11/750. Avalon supplies the compute-bound UNIX utilities such as nroff, troff, and the C compiler. Applications may be run on the Avalon system after a simple recompilation of the source.

COMPANY NAME: BRS
ADDRESS: 1200 Route & Latham
New York, NY 12110
TELEPHONE NUMBER: (518) 783-1161
MARKETING CONTACT: Richard C. Simon
(703) 527-4503

BRS/SEARCH is a text storage and retrieval system that permits database creation, loading, maintenance, and full-text/free-text searching. The system currently operates on hardware from seven companies under the UNIX operating system. Applications include litigation files, office correspondence, engineering records and documents, personnel files, library files, and many others. BRS/SEARCH permits retrieval on any word, parts of words, and combinations using Boolean, positional, and relational operators. There is no limit to the length of records or fields. The system has been used to store the full text of textbooks and encyclopedias. Dealer and OEM inquiries are invited.

COMPANY NAME: Benjamin Cummings
Publishing Company
ADDRESS: 2727 Sand Hill Road
Menlo Park, CA 94025
TELEPHONE NUMBER: (800) 227-1936
MARKETING CONTACT: Jake Warde

Benjamin Cummings publishes an outstanding list of UNIX, Xenix, System-5 and C books. The following titles are now or' will be available soon:

- *A Practical Guide to UNIX* by M. Sobell - published 1984
- *A Book on C* by Kelley and Pohl - April 1984
- *A Practical Guide to Xenix* by Sobell - June 1984
- *A Practical Guide to System 5* by Sobell - July 1984

Books can be ordered by calling: (800) 227-1936.

COMPANY NAME: Britton-Lee. Inc.
ADDRESS: 14600 Winchester Boulevard
Los Gatos, CA 95030
TELEPHONE NUMBER: (408) 378-7000
MARKETING CONTACT: Kathryn Scollon

Britton-Lee's IDM (Intelligent Database Machine) is a Relational Database Manager implemented in a dedicated function processor. The IDM achieves high performance (5-10 times faster than software-implemented Relational DBMS) by offloading the DBMS to special-purpose "backend" hardware. The IDM can function as a shared dataserver for two or more different host CPUs. Supported hosts include:
IBM PC/PC-DOS
VAX/UNIX (Berkeley 4.2 BSD)
PDP-11/UNIX (Version 7.0)
VAX/VMS
Host-support software provides interactive keyboard queries, queries embedded in applications programs and extensive subroutine library extensions.

COMPANY NAME: CGA Computer, Inc.
ADDRESS: 1011 East Touhy Avenue
Des Plaines, IL 60018
TELEPHONE NUMBER: (312) 296-9660
MARKETING CONTACT: Tom Drew

CGA is one of this country's largest full-service data processing consulting firms. We are a fifteen-year-old firm based in Halmdel, NJ and employ 600 full time consultants nationwide in 17 offices in 14 states. One of our largest clients is AT&T Bell Laboratories, where we employ 80 UNIX and C professionals. We are a publically held OTC (NASDAQ) with sales projected at $40 million for the current fiscal year. CGA offers a wide variety of DP-related services from feasibility studies to systems design and analysis to contract programming and technical skills training. CGA most recently entered the software marketplace with many products for the IBM environment.

COMPANY NAME: CMI Corporation
ADDRESS: 755 West Big Beaver Road, Suite 1900
Troy, MI 48084
TELEPHONE NUMBER: (313) 362-1000
MARKETING CONTACT: Dennis J. Pennington, V.P.

CMI Corporation is displaying an IBM Series 1 running under SERIX, a full implementation of System III UNIX. Products complimenting SERIX are an enhanced 'C' compiler, a Visual Editor, a 'C' compiler under EDX, R/M Cobol, the Unify data base and ViewComp spreadsheet accounting. Products soon to be available are an EDL to 'C' converter and a word processing package.

COMPANY NAME: CADMUS Computer Systems, Inc.
ADDRESS: 600 Suffolk Street
Lowell, MA 01854
TELEPHONE NUMBER: (617) 453-2899
MARKETING CONTACT: David Schell, Marketing Support Manager

CADMUS 9000 systems are 32-bit multipurpose computers that deliver the power of a mainframe over a distributed network environment. Designed from the bottom up as a networked node, each CADMUS system provides users with a dedicated MC68010 microprocessor, an extensively enhanced multiuser UNIX Operating System, Q-Bus and Multibus peripheral compatibility, virtual memory and high-resolution color graphics.

Available in multiple configurations, ranging from diskless workstations to high-capacity multiuser networks, CADMUS systems are networked over Ethernet or 50M bit/sec Fiber Optic Net, into a shared-resource system environment of unprecedented capabilities. UNISON, the CADMUS distributed network software, makes full, transparent access to the entire networks processing resources available to each individual node.

COMPANY NAME: Callan Data Systems
ADDRESS: 2645 Townsgate Road
Westlake Village, CA 91361
TELEPHONE NUMBER: (818) 991-9156
MARKETING CONTACT: Bob Hufnagel, V.P. Marketing

Callan Data Systems demonstrated the UNISTAR family of Multibus super microcomputers running UNIX System V. The UNISTAR 100 singleuser and UNISTAR 200 multiuser are 8-slot 68000-based systems with up to 43 Mbyte disks. The UNISTAR 300 is a 12-slot 68010 10-MHz system with 2 Mbytes of RAM, 172-Mbyte disk and 45-Mbyte tape backup. Languages supported include C Compiler, FORTRAN 77, Pascal, Ada, COBOL, and BASIC. Options include floating point accelerator, array processor, intelligent controller, TCP/IP protocol and networking. Optional application software offered are EasyType word processing, UNICALC spreadsheet, MicroINGRES relational data base management system.

COMPANY NAME: Cambridge Digital Systems
ADDRESS: P.O. Box 568
65 Bent Street
Cambridge, MA 02139
TELEPHONE NUMBER: (617) 491-2700,
(800) 343-5504
MARKETING CONTACT: Dena Wortzel

Cambridge Digital Systems is an integrator of customized DEC, NCR and UNIX compatible systems and software. The company's Uniforum exhibit focused on its new 68000/68010-based Univax CPU, designed to optimize UNIX performance. On display was the Univax configuration of CDS's pre-packaged Q-bus-based System 94 line, running a true adaptation of System III, System V and Berkeley 4.2 UNIX with a 70-Mbyte Winchester and ¼-inch streaming tape. The Univax System 94 is a compact 12-user system with a 10½-inch desktop, rack-, or floor-mountable enclosure. The East Coast distributor of the NCR Tower 1632, CDS exhibited this 16-user system running an adaptation of UNIX System III.

COMPANY NAME: Charles River Data Systems
ADDRESS: 983 Concord Street
Framingham, MA 01701
TELEPHONE NUMBER: (617) 626-1000
MARKETING CONTACT: Jim Isaak

Charles River offers the Universe 68 line of OEM supermicro computers—designed for high performance and low cost (over 1 MIPS for under $10,000). Running UNIX System V or the UNOS operating system with its real time and high reliability extensions. Configurations range from 10MB disk with 512KB RAM and four users to 5+MB RAM, 200+MB disk and up to 64 serial ports. The systems are built on 12.5 MHz 68000 main processors with cache and a second 68000 processor for character device control. Floating point, array processors and graphics interfaces are also available.

COMPANY NAME: CIFER Inc./C&P Int'l.
Consultants, Inc.
ADDRESS: 375 North Broadway, Suite 302
Jericho, NY 11753
TELEPHONE NUMBER: (516) 935-8490
MARKETING CONTACT: Gene Bidwell

CIFER, a British manufacturer of terminals and microcomputers that feature advanced multiprocessor architecture and high-resolution graphics, introduces its new UNIX product line of Personal Workstations, Personal Microcomputers, and Multi-User Microcomputers. CIFER UNIX machines contain the powerful 16-bit MC68000 with Unisoft's Uniplus version of Bell Laboratories System III UNIX Operating System. All have a built-in Winchester, 800KB floppy disk, RS232 serial interfaces, Centronics parallel interface, IEEE 488 bus, external floppy disk bus, and a Z80A 128K processor with CP/M Plus Operating System. All CIFER workstations contain an intelligent VDU on a separate Z80A processor and provide a rich selection of transmission and display functions.

COMPANY NAME: Compugraphic Corporation
ADDRESS: 200 Ballardvale Street
Wilmington, MA 01887
TELEPHONE NUMBER: (617) 658-5600 x5087
MARKETING CONTACT: Carl E. Miller,
Manager, OEM Sales

We displayed a Compugraphic 8400 digital typesetter being driven directly by a modified TROFF program prepared by Technical Type and Composition of Salem, OR. During the show we ran this TROFF program on a Pixel; however, this software is portable to most UNIX-based systems. The use of this software to drive the 8400 will allow a company to directly typeset their documentation. Compugraphic Corporation is a supplier of typesetting equipment in the low- to medium-price ranges and is making an effort to be as compatible as possible with TROFF and UNIX-based systems.

COMPANY NAME: Computer Consoles, Inc.
ADDRESS: 97 Humboldt Street
Rochester, NY 14609
TELEPHONE NUMBER: 1-(800) 833-7477
MARKETING CONTACT: Edward W. Scott, Jr.
(703) 471-6860

OFFICEPOWER™ is a multifunctional, highly integrated office automation system providing turnkey applications in support of general office automation functions, plus a relational database management system, a user-defined forms package, a query language and a report generator.

OFFICEPOWER runs on the Power 5/20 standalone, one of CCI's family of MC68000-based computers produced by CCI. The Power 5/20 can be expanded into multiprocessor systems that support a network of terminals and are designed to incorporate CCI's fault-tolerant "PerpetualProcessing™" system.

COMPANY NAME: Computer Services Corporation
ADDRESS: 2-6 Nishi Shinjuku, Shinjuku
Tokyo, JAPAN 160-91
TELEPHONE NUMBER: (701) 569-6300
MARKETING CONTACT: Mr. M. Fujita (Computer Services International)

Terranet is an inexpensive local area network. Tap units use RS-232C to connect to various forms of data equipment (e.g. terminals, modems, printers and computers). Data is sent to other tap units along a coaxial cable which can extend up to 500m. Any tap can connect to many machines without running many cables.

Time division multiplexing is used to provide thirty 9600-baud, noncontending, virtual circuits. Switch-selectable firmware within each tap can provide a conversational dialogue at user terminals, short status codes at computer connections or gateway actions when taps are used to connect two networks.

COMPANY NAME: Computer Technology Group Telemedia, Inc.
ADDRESS: 310 South Michigan Avenue
Chicago, IL 60604
TELEPHONE NUMBER: (312) 987-4082
MARKETING CONTACT: Gary Slavin

The Computer Technology Group specializes in UNIX and C training given in public and on-site seminars, as well as offering video-based and interactive videodisc training courses. CTG's video-based training integrates professionally developed and produced video and text materials, including hands-on exercises. Our courses are produced with the highest standards of video quality applying the latest techniques of instruction design including the use of computer graphics and animation, compressing learning time.

CTG's interactive videodisc system, co-developed with Interactive Training Systems, Inc. contains computer generated text and graphics designed to direct students and monitor their progress through CTG's video training materials.

COMPANY NAME: Computerized Office Services, Inc. (COSI)
ADDRESS: 313 North First Street
Ann Arbor, MI 48103
TELEPHONE NUMBER: (313) 665-8778
MARKETING CONTACT: Keith Burleigh

COSI is a full-service UNIX software house dedicated to serving the commercial data processing industry. At UniForum '84, we introduced two new product lines:

- VISUAL—visually-oriented product development tools for commercial data processing
- LINK—communications packages linking personal computers, UNIX machines, and mainframes.

COSI also provides UNIX software services such as:

- System design of complex UNIX-based corporate information systems
- Conversion of software, both custom and product, to a UNIX environment
- Educational assistance for DP staffs adapting to a UNIX culture.

COMPANY NAME: Computerware, Inc.
ADDRESS: 8480-I Tyco Road
Vienna, VA 22180
TELEPHONE NUMBER: (703) 821-8200
MARKETING CONTACT: John Dargo

Computerware, Inc. distributes a full range of computer supplies, furniture and peripheral equipment. They are also the authorized Virginia dealer for the Texas Instruments Professional Computer. Featured at the UniForum Conference were the Ambassador and Guru video display terminals manufactured by Ann Arbor Terminals, Inc. These multi-featured terminals are favorites among UNIX users.

COMPANY NAME: Conetic Systems, Inc.
ADDRESS: 12025 N.E. Marx, Suite 5H
Portland, OR 97220
TELEPHONE NUMBER: (503) 253-0074
MARKETING CONTACT: R. Richard Schwindt

Conetic Systems markets a data base and application generation system written in C and fully integrated accounting modules developed with the application generation system. The product runs identically under UNIX and MS/DOS. Multiple utilities have been written to support virtually all types of data management including relationships. All data types and file structures including hashed, ISAM, and sequential files. Files can be loaded to spread sheets or other programs or accepted from other data bases. The product is presently used by Fortune 500 companies, including NCR and Allied Corporation, with more general distribution now in progress.

COMPANY NAME: Cucumber Bookshop, Inc.
ADDRESS: 5611 Kraft Drive
Rockville, MD 20852
TELEPHONE NUMBER: (301) 881-2722
MARKETING CONTACT: Dave Harris

UNIX and C Language book display including all books currently available, and catalog of forthcoming books due in the next 3 to 6 months; the most complete list of UNIX books, as far as we know.

Also, demonstrating an information retrieval system, SIRE™ running on UNIX, with automatic full-text indexing, ranked output of usefulness, word stem matching, related term searching, variable length records, and more.

COMPANY NAME: Data General Corporation
ADDRESS: 4400 Computer Drive G-134
Westboro, MA 01580
TELEPHONE NUMBER: (617) 366-8911 x5281
MARKETING CONTACT: Barbara Loonam

Data General's UNX/VS is a UNIX operating system environment for the 32-bit ECLIPSE MV/Family Systems. It is fully integrated with DG's AOS/VS operating system, thereby providing a dual environment for the user. All AOS/VS applications, abilities, languages, debuggers and communication facilities may be accessed from UNX/VS.

The UNX/VS operating system environment is based on AT&T's System V UNIX and includes Berkeley enhancements, i.e. C shell, VI and termcap.

COMPANY NAME: Dual Systems Control Corporation
ADDRESS: 2530 San Pablo Avenue
Berkeley, CA 94702
TELEPHONE NUMBER: (415) 549-3854
MARKETING CONTACT: Joan R. Stivers, Marketing Director

Dual produces highly-reliable, exceptionally high-performance supermicro systems that meet the industrial standards of professional engineers. Dual first offered the MC68000 with UNIX in 1981 and has established a solid reputation for delivering stable, proven products. Dual delivers several major technical advantages over competing 68000-based systems including DMA on serial RS-232C outputs, 256B FIFO buffer on serial inputs, and an SMD disk controller that supports up to 1.2GB of Winchester disk and provides the highest throughput on a UNIX system. Also available on these 12- to 16-user systems are floating point processing, up to 3.25MB of RAM, 9-track tape drive and interface, graphics boards, Fortran 77, Pascal, RM/Cobol, Basic, Lisp, Real Time Kernel, Lex word processing, MBSI accounting, Viewcomp spreadsheet, Micro Ingres, Unify; and, of course, System V with Berkeley enhancements.

COMPANY NAME: ELXSI
ADDRESS: 2334 Lundy Place
San Jose, CA 95131
TELEPHONE NUMBER: (408) 942-0900
MARKETING CONTACT: Dave Reed

The ELXSI System 6400 is a true 64-bit multiprocessor supercomputer running System V. It consists of a 320 Mbyte/second central bus allowing smooth growth from 1 to 10 CPUs, 8 to 192 Mbytes of main memory, 16 to 64 Mbytes/second I/O capacity. CPUs are built using proprietary ECL chips and run 4 million Whetstone instructions/second EACH, for system speeds from 4 to 40 MIPS. Processes each have up to 4 GigaBytes of virtual address space. Languages: C, Pascal, FORTRAN-77, BASIC, COBOL-74, Main Sail. High-performance INGRES-based relational database. Outstanding price/performance ratio.

COMPANY NAME: Excelan, Inc.
ADDRESS: 2180 Fortune Drive
San Jose, CA 95131
TELEX: 176610
TELEPHONE NUMBER: (408) 945-9526
MARKETING CONTACT: Doug Planchon

"UNIX on Ethernet with computer Inter-operability" was the theme of Excelan's booth at the 1984 Uniforum Conference. The Excelan Open Systems (EXOS™) concept was demonstrated with the integration of four different UNIX-based computers, each from a different manufacturer designed for applications from general business to factory automation and CAD/CAM, each sharing the resources of the other. This was made possible by Excelan's EXOS 101 Ethernet front-end processor boards and onboard EXOS 8010 TCP/IP protocol software which are independent of host CPU type, UNIX version or source of the port. Also shown was the Excelan Nutcracker™, the world's only analyzer/simulator for Ethernet systems.

COMPANY NAME: Federal Marketing Group, Inc.
ADDRESS: 100 West Hill Street
Baltimore, MD 21230
TELEPHONE NUMBER: (301) 539-2121
MARKETING CONTACT: David E. Fedder or
Elizabeth A. Fedder

Federal Marketing Group, Inc. is a distributor offering rapid availability of UNIX- and MS/DOS-based hardware and software at aggressive prices coupled with strong systems engineering support. Inquiries are invited from dealers, systems houses, government agencies, educational institutions, and large corporations. Lines offered include Altos 68000, 586, and 986; APPGEN program generator and accounting software from Software Express; R Word; Horizon integrated office automation; Texas Instruments Professional and Portable PC; Avatar PC/3278 computer and terminal; Samna word processing; Executec SeriesOne; Mosaic Integrated 6 with graphics; and other superior products. Dealers are eligible for free training in UNIX!

COMPANY NAME: Fortune Systems
ADDRESS: 101 Twin Dolphin Drive
Redwood City, CA 94065
TELEPHONE NUMBER: (415) 595-8444
MARKETING CONTACT: Carolyn Carnifix

Fortune Systems has over 30,000 workstations installed worldwide, currently the largest installed base of UNIX™ systems. Fortune Systems' multiple-user microcomputer is designed for both small and large business with such applications as: word processing, spreadsheet, business accounting applications, Fortran, Cobol, C, and Basic, plus a wide variety of third-party software. Fortune offers complete support, training and maintenance programs.

COMPANY NAME: Government Computer News
ADDRESS: 1620 Elton Road
Silver Spring, MD 20903
TELEPHONE NUMBER: (301) 445-4405
MARKETING CONTACT: David Ross

Government Computer News is a monthly publication dedicated to the interests and needs of computer managers and professionals, and end-users of computer products and services throughout the government.

COMPANY NAME: Handle Corporation
ADDRESS: P.O. Box 7018
Tahoe City, CA 95730
TELEPHONE NUMBER: (916) 583-7283
MARKETING CONTACT: Dianne Harlow,
Director of Marketing
Communications

The Handle family of products is a state-of-the-art, interactive line of software for office automation, which runs under the UNIX operating system. All Handle products are completely function-key driven and offer such unique features as: archiving, proof mode, and in-text graphics. The Handle family of products includes: Handle Writer (word processor), Handle Spell, Handle Graphics, Handle Calc, Handle List (database management system), and Handle Access (access to mainframe database).

COMPANY NAME: Hayden Book Company
ADDRESS: 10 Mulholland Drive
Hasbrouck Heights, NJ 07604
TELEPHONE NUMBER: (201) 393-6000
MARKETING CONTACT: Barbara C. Garris

Hayden Book Company has begun distribution of AT&T Bell Labs UNIX system videotapes. They explain and discuss the UNIX operating system, the most portable and expandable operating system available in today's computer environment.

The tapes provide visual explanation and discussion of the UNIX System V, it's concept, operation and applications. Explanation is provided by the creators and pioneers of the UNIX system, including Kenneth Thompson, Dennis Ritchie, and Brian Kernighan. The tapes, produced by Belove-Laiserin are available in both Beta and VHS formats.

COMPANY NAME: Hewlett-Packard Corporation
ADDRESS: 19447 Pruneridge Avenue
Cupertino, CA 95014
TELEPHONE NUMBER: (408) 725-8111
MARKETING CONTACT: Doug Hartman

HP offers a range of high-quality, high-performance systems which run HP-UX, HP's enhanced version of System III UNIX. Products featured at the show included engineering workstations and multiuser systems. All systems highlighted graphics capabilities.

COMPANY NAME: Human Computing Resources
ADDRESS: 10 St. Mary Street
Toronto, Ontario M4Y 1P9
CANADA
TELEPHONE NUMBER: (416) 922-1937
MARKETING CONTACT: Jim Peters

Human Computing Resources exhibited Chronicle™, its new line of advanced business applications software especially created for the UNIX operating system. Chronicle is particularly well suited to 16- and 32-microprocessor systems. It may be run on almost any machine using UNIX and is portable to many major relational databases. As well, HCR displayed its UNITY™ operating system for a NS16032-based machine and VAX UNITY™ under VMS™, its emulation of UNIX giving VMS users access to both systems in a correct and non-privileged manner.

COMPANY NAME: IBC/Integrated Business Computers
ADDRESS: 21621 Nordhoff Street
Chatsworth, CA 91311
TELEPHONE NUMBER: (818) 882-9007
MARKETING CONTACT: Randy Rogers

IBC displayed the IBC ENSIGN supermicro. The ENSIGN is an extremely fast multiuser UNIX System III system. The ENSIGN supports 32 users with up to 8 Mbyte of memory and over 1000 Mbyte of disk storage.

In late first quarter, the ENSIGN will support UNIX System V with the MC 68010. The ENSIGN was designed to run multiuser UNIX faster than any other supermicro.

COMPANY NAME: Information Industries, Inc.
ADDRESS: 8880 Ward Parkway
Kansas City, MO 64114
TELEPHONE NUMBER: 1-(800) TRIPLE I
(In Missouri:
(816) 444-8100
MARKETING CONTACT: David Francis

Information Industries, Inc. (Triple-I) provides a flexible, risk-free alternative to traditional hiring and recruiting methods. This alternative, called Skills Management, allows Fortune 500 companies to overcome problems like headcount limitations, peak load demands, and high recruiting costs. The result is increased productivity and cost-efficiency.

Triple-I's Skills Management Program delivers these benefits by providing highly skilled data processing/engineering consulting professionals who are experts in areas like networking, UNIX, C, M-204 and OS/MVS, to name a few. Through constant exposure to state-of-the-art environments and career planning techniques, these Triple-I professionals are able to greatly enhance their career opportunities and potential.

COMPANY NAME: Integrated Computer Systems
ADDRESS: 6305 Arizona Place
Los Angeles, CA 90045
TELEPHONE NUMBER: (213) 417-8888
MARKETING CONTACT: Carolyn Yost

Integrated Computer Systems is devoted to providing unbiased, practical education in advanced technology. The courses provide engineers, system designers and their managers with a solid technical background and up-to-date information on the state-of-the-art in applying technology. Formats include 4-day short courses, a self-study program, video courses and custom designed on-sites.

Subjects included are: networks and systems, man/machine systems, digital processing, minis and micros, software and management courses.

COMPANY NAME: Interlan, Inc.
TELEPHONE NUMBER: (617) 692-3900
MARKETING CONTACT: Gerald W. Wesel

Interlan, Inc. will demonstrate their two newest NET/PLUS™ products; the NTS10 Network Terminal Server, a four- or eight-port unit that interfaces RS232-C devices onto the Ethernet LAN, and Multi-Vendor Personal Computer Networking Software which provide file transfer and terminal emulation for over a dozen personal computers. When used in conjunction with the Network Terminal Server, this software provides communications between PC-to-Host, PC-to-PC, and PC-to-Device.

Interlan will also demonstrate a complete Ethernet data communications package linking together host computers running UNIX to other data processing equipment on the Ethernet network. Demonstrations will include remote log-in, file transfer, and connection to other communication services.

COMPANY NAME: Logical Software, Inc.
ADDRESS: 55 Wheeler Street
Cambridge, MA 02138
TELEPHONE NUMBER: (617) 864-0137
MARKETING CONTACT: Ilona A. Lappo

SoftShell is a full-screen user interface to UNIX with a screen formatter for building a forms or menu interface to applications programs. SoftShell provides command templates for all major UNIX commands, reverse scrolling with full-screen editing and re-execution of commands, as well as facilities for displaying the many tree structures of UNIX, such as directory and file systems.

LOGIX is a relational database management system that is completely integrated with the UNIX operating system. LOGIX runs at the shell level, allowing smooth alternation between UNIX and LOGIX commands and allowing LOGIX commands to be packaged in UNIX shell scripts. LOGIX includes a complete interactive command language, a full-screen relation editor, Q programming language/report writer, query compiler, and C-interface. A history of all changes to a relation can be maintained through the use of dated relations and several databases can be accessed simultaneously.

COMPANY NAME: MASSCOMP
ADDRESS: One Technology Park
Westford, MA 01886
TELEPHONE NUMBER: (617) 692-6200
MARKETING CONTACT: Catherine Pfister

MASSCOMP designs, manufactures, sells, and services high-performance, multi-user, M68010/M68000 UNIX™ systems to End-Users and OEMs in engineering, science, and education. The MC-500 and MC-500 WorkStation are based on a triple-bus architecture which anticipates future technological advances in microprocessors, memory, mass-storage, communications, and real-time interfaces. The CPU utilizes a proprietary bus connecting the M68010/M68000 CPU to system memory, and floating point and array processors. A performance-enhanced Intel Multibus™ supports multiple Independent Graphics Processors and a wide selection of system peripherals, including an Intelligent Ethernet™ controller board. For data acquisition, the MC-500 supports the Data Acquisition/Control Processor which controls an enhanced STD Bus for high-speed data acquisition and control at rates up to 1 megasample per second. The Real-Time UNIX operating system is based on UNIX System III, Berkeley virtual memory and Ethernet enhancements, and MASSCOMP's Real-Time and QUICK-CHOICE™ User Interface enhancements.

COMPANY NAME: Megadata Corporation
ADDRESS: 35 Orville Drive
TELEPHONE NUMBER: (516) 589-6800
MARKETING CONTACT: Richard Adams

Megadata's Series 8000 terminals and systems are designed around the 68000 microprocessor.

The 8174 Model 1 and Model 4 are designed utilizing Multibus™ compatible modules and include UniPlus⁺, a UNIX™ operating system with real-time enhancements and a choice of programming languages.

The Model 8174 Model 3 controller supports up to 32 Model 8178 terminals over a high-performance 1M bit/second twinax link. Terminals can be located up to 2000 feet from the controller.

The 8178 terminal offers display height adjustment and rotation. Screen tilt, and keyboard angle are also adjustable. The programmable keyboard with up to 155 keys is provided for any application requirement.

COMPANY NAME: Micro Focus, Inc.
ADDRESS: 2465 East Bayshore Road
Suite 400
Palo Alto, CA 94303
TELEPHONE NUMBER: (415) 856-4161
MARKETING CONTACT: Dan Fineberg
Marketing
Communications
Manager

Micro Focus is the leading producer of COBOL compilers and programmer productivity tools for business application development. Micro Focus products are available for a wide variety of 8-, 16- and 32-bit microprocessor-based computers, and they support dozens of standard and proprietary operating systems including UNIX V7, BSD 4.1, XENIX, System III and System V. High-performance LEVEL II COBOL is written in the native code of the CPU on which it operates and includes a native code generator for fast program execution. Like Compact LEVEL II COBOL, the compiler is GSA certified to the Federal High Level of the ANSI '74 COBOL standard. FORMS-2 is a user-friendly source code generator for creating interactive screen displays. ANIMATOR is a source code-level interactive debugging and program analysis tool. Together, the products provide an accessible and interactive alternative to developing applications on the mainframe.

COMPANY NAME: Microsoft Corporation
ADDRESS: 10700 Northup Way
Bellevue, WA 98004
TELEPHONE NUMBER: (206) 828-8080
MARKETING CONTACT: John Ulett
Xenix Product Marketing
Manager

Microsoft is the developer of Xenix, the most widely used licensed version of UNIX on microcomputers. Xenix is specifically designed for the commercial marketplace, and has numerous enhancements such as usability and reliability, added functionality, increased system security, MS-DOS compatability, system administration utilities, and improved documentation. Xenix is consistent across major microprocessors, including Intel's 8086 and 286, Motorola's 68000, National Semiconductor's 16032, and Zilog's Z8000. There are more commercial applications available for Xenix than any other form of UNIX. Microsoft is demonstrating Xenix on the Intel 310, the Durango Poppy, the Radio Shack Model 16, and the IBM PC-XT.

COMPANY NAME: Momentum Computer Systems
International
ADDRESS: 2730 Junction Avenue
San Jose, CA 95134
TELEPHONE NUMBER: (408) 942-0638
MARKETING CONTACT: Albert R. Pfeltz
National Sales Manager

Momentum manufactures and markets an entire family of 68000, UNIX-based super-microprocessors. The 32/4 is a second generation professional workstation with 1MB of memory, 4 serial ports, a graphics subsystem with a second 68000 and ¼MB of memory, a 15-in. monitor (1024 x 704), two 5MB removable cartridge Winchesters and detachable keyboard. The 32 and 32/E are 2- to 16-user systems with 1MB of memory, floppy or streaming tape, one or two 5¼-in. Winchester drives, and parallel port. Software available includes UNIX System III, RM/Cobol, Cogen, SVS Pascal, SVS Fortran, SMC Basic, C, 68000 assembler, word processing, spreadsheet, and MBSI accounting packages.

COMPANY NAME: NCR Corporation
ADDRESS: 3325 Platt Springs Road
TELEPHONE NUMBER: (803) 796-9250
MARKETING CONTACT: M. D. Lambert

NCR TOWER 1632.

1. Distributed Resource System of TWIN TOWERS.
2. UNIX Version 7 with application tools:
   - WORDMARC Office Automation
   - Ingres Relational DataBase
   - Multiplan
   - Color Graphics
3. UNIX System 5 with TOWERNET linked to other booths at the conference.
4. A preview of Power-Fail Recovery in the TOWER in the event of power outage.

COMPANY NAME: National Semiconductor Corporation
ADDRESS: 2900 Semiconductor Drive
Santa Clara, CA 95051
TELEPHONE NUMBER: (408) 721-5000
MARKETING CONTACT: Dave Vonasek

UNIX products include the SYS16™, an 8-user development system that utilizes the GENIX™ operating system. The GENIX O.S., an enhancement of Berkeley 4.1 UNIX, is also available in source form as well as binary. Additional products include cross software for the NS16000 family on VAX, in-system emulators, and 8-bit development systems.

COMPANY NAME: Netcom Products
ADDRESS: 430 Toyama Drive
Sunnyvale, CA 94089
TELEPHONE NUMBER: (408) 744-0721
MARKETING CONTACT: Tom McKee

Fully integrated microcomputer systems featuring:
- MC68000 CPU
- Multibus
- UNIX

A full complement of peripherals and options:
- 20-40MB Winchester drives
- 84-474MB SMD
- ½-in. magnetic tape
- 50-MB cartridge storage (25 fixed)
- Floating point
- Rack mount, tower, cabinet enclosures

COMPANY NAME: Network Research Corporation
ADDRESS: 1964 Westwood Boulevard
Suite 200
Los Angeles, CA 90025
TELEPHONE NUMBER: (213) 474-7717
MARKETING CONTACT: Kyle Todd

FUSION™ ethernet software is capable of interconnecting IBM-PC, VAX, 68000, PDP-11 and 8086 processors on the same LAN. FUSION is able to cross over operating system boundaries, linking machines running UNIX, MS-DOS and VMS. FUSION provides the user with the ability to transfer files between systems, to execute commands on a remote host, or establish a virtual terminal where, for example, an IBM-PC under MS-DOS or UNIX can act as a terminal to a VAX running UNIX or VMS.

COMPANY NAME: Officesmiths, Inc.
ADDRESS: 331 Cooper Street
Ottawa, Ontario K2P 0G5
TELEPHONE NUMBER: (613) 235-6749
MARKETING CONTACT: Glenn A. McInnes, President

The Officesmith™ is a document management system designed to provide organizations with a solution to the maintenance of information. The Officesmith operates in a structured environment that includes multiple windows, menus, single function key commands and a user-programmable help facility, and supports a complete range of document management activities. Combining powerful yet simple application development tools and document management facilities, The Officesmith offers systems integrators and office systems developers the resources to build and maintain responsive corporate information structures. OfficePolicy is a methodology for documenting, communicating and accessing policies and procedures in large organizations. OfficePolicy is a complete approach combining management guides, training courses, and software tools based on The Officesmith.

COMPANY NAME: Onyx Systems, Inc.
ADDRESS: 25 E. Trimble Road
San Jose, CA 95131
TELEPHONE NUMBER: (408) 946-6330
MARKETING CONTACT: Tom Anthony
VP Marketing

Onyx Systems exhibited UNIX-based microcomputer systems first introduced at Comdex Fall. The model C5012D is a desktop system with 512K bytes of RAM, 14- or 21-Mbyte Winchester disk and tape cartridge backup. It supports five users under UNIX System III. The C5012V is a floor-standing unit with up to 1Mbyte of memory and up to 42 Mbytes of Winchester storage. The systems are bundled with the Onyx Office, an integrated applications package including word processing, spreadsheet and database manager.

COMPANY NAME: Oregon Software, Inc.
ADDRESS: 2340 SW Canyon Road
Portland, OR 97201
TELEPHONE NUMBER: (503) 226-7760
MARKETING CONTACT: David Cloutuer

Oregon Software exhibited their Pascal-2 compiler system. This includes a high-performance compiler for the MC68000 and PDP-11 running on UNIX. A source-level, interactive debugger, performance profiler, and utility package are also included.

The MC68000 UNIX compiler executes on the Unisoft, Uniplus+ system and Xenix. End-use pricing for Pascal-2 on the MC68000 is $1,650 including one year of support. OEM prices available upon request.

COMPANY NAME: Pacific Microcomputers, Inc.
ADDRESS: 119 Aberdeen Drive
Cardiff, CA 92007
TELEPHONE NUMBER: (619) 436-8649
MARKETING CONTACT: Sherrell Harper

Pacific Microcomputers' display consisted of several levels of 68000/68010 single-board microprocessors displaying the latest technology in controlling multiprocessor environments, high-speed data transfer, and no-wait-state processing for OEM applications. In addition, Pacific exhibited fully integrated microprocessors with UNIX operating system and full support of all standard programming languages. These systems are highly flexible designs utilizing both 5¼-in. and 8-in. technology providing high performance aimed at the multiuser marketplace for OEMs and sophisticated end users.

COMPANY NAME: Paradyne Corporation
ADDRESS: PO Box 2826
8550 Ulmerton Road
Largo, FL 33540
TELEPHONE NUMBER: (813) 530-2577
MARKETING CONTACT: Elliott Kane

Paradyne's exhibit included demonstrations and presentations of a UNIX-derived communications Terminal Controller. Supporting multiple protocols and multiple terminal personalities, two models were displayed. A desktop System 8400 supports up to 8 terminals and or printers with 4 communications ports and a 26MB disk and 640KB diskette. The application development System 8400 supports up to 16 terminals and or printers with a 9-track 1600-bpi tape and 160MB disk. Both systems use a UNIX-derived operating system and a combination of microprocessors—the Z8000 and the 80186 for the CPU and the communications controller, respectively. Paradyne is a communications corporation and has installed over 1900 System 8400s.

COMPANY NAME: PHACT Associates, Ltd.
ADDRESS: 225 Lafayette Street
New York City, NY 10012
TELEPHONE NUMBER: (212) 219-3744
MARKETING CONTACT: David Graham

PHACT Associates provide a range of software tools aimed at the applications builder and end-user.

Products include: PHACT-dbrm, a multi-keyed ISAM-like database record manager, incorporating a data dictionary, variable-length records, etc.; PHACT-rql, a relational query and data manipulation language; PHACT-rg, a report generator.

New products include SaPHACT, a UNIX filter for manipulating PHACT databases. Products under development include a List Manager and an Object Manager.

COMPANY NAME: Plexus Computers, Inc.
ADDRESS: 2230 Martin Avenue
Santa Clara, CA 95050
TELEPHONE NUMBER: (408) 988-1755
MARKETING CONTACT: Brenda Birrell

Plexus initiated its "Take the Plexus Challenge" program and demonstrated a local area network including the Plexus P/65 network file server, two IBM PCs as UNIX-based workstations, and a Plexus P/35 32-bit supermicrocomputer.

Under the "Take the Plexus Challenge" program, attendees ran benchmark studies on a Plexus P/35.

The most recent Plexus computer developed, the P/65, offers up to 1.1 Gigabytes of storage and serves as a database management and network management tool. In combination with the UNIFY DBMS, it provides fully transparent multiuser access to shared data on an Ethernet-based network. IBM PCs were linked to the network as standalones or UNIX-based workstations through Lantech Systems' uNETix software.

COMPANY NAME: Productivity Products International
ADDRESS: Newton, CT 06482
TELEPHONE NUMBER: (203) 426-1875
MARKETING CONTACT: Mr. Andrew Iorio

Productivity Products has a product named Objective-C. This product brings the advantages of object-oriented programming to the C language. We are producing tool kits for professional programmers and coordination tools for systems houses.

COMPANY NAME: Proper Software
ADDRESS: 2000 Center Street
Suite 1024
Berkeley, CA 94704
TELEPHONE NUMBER: (415) 540-5958
MARKETING CONTACT: Paul Hoffman

We sell DRIVER, a visual replacement for the CD command, to OEMs.

COMPANY NAME: Quadratron Systems, Inc.
ADDRESS: 15760 Ventura Boulevard, #1032
Encino, CA 91436
TELEPHONE NUMBER: (818) 789-8588
MARKETING CONTACT: Vanessa Abbe

Q-Office is an office automation software package that includes Q-One word processing; Q-Date electronic calendar; Q-Menu menu generator; Q-Note electronic notepad; Q-Form form generator, Q-Call phone directory; Q-Math calculator; and Q-Mail electronic mail. Q-Office is completely device independent and comes with termcap and printcap files and generators.

COMPANY NAME: Relational Technology, Inc.
ADDRESS: 2855 Telegraph Avenue
Berkeley, CA 94705
TELEPHONE NUMBER: (415) 845-1700
MARKETING CONTACT: Peter Tierney
Vice President Marketing

Relational Technology, Inc. displayed INGRES, a relational database management and applications development system. INGRES provides flexible report writing capabilities; data management and applications generation through Visual Programming™ forms interfaces, and a powerful query language, QUEL, for interactive use or easy embedding in most programming languages. INGRES includes INGRES/REPORTS™; INGRES/QUERY™; INGRES/FORMS™; INGRES/APPLICATIONS-BY-FORMS™; INGRES/GRAPHICS™; INGRES/NET™ for distributed access; and EQUEL (Embedded QUEL) interfaces for FORTRAN, COBOL, BASIC, PASCAL, and C languages. INGRES runs on VAX computers under the UNIX or VMS operating system, or on MC68000-based supermicrocomputers under UNIX operating systems.

COMPANY NAME: STSC, Inc.
ADDRESS: 2115 E. Jefferson Street
Rockville, MD 20852
TELEPHONE NUMBER: (301) 984-5000
MARKETING CONTACT: Rich Paulson

STSC has developed a high-performance enhanced APL interpreter for the UNIX environment, to complement its very popular APL*PLUS/PC system and its APL products for IBM mainframes. The APL*PLUS/UNIX system runs on a wide variety of supermicro and minicomputers that use UNIX System 3 or System V. This all-new APL system provides features that previously have been available only on mainframe systems—including nested arrays and very large APL variables and workspaces.

COMPANY NAME: Sage Computer
ADDRESS: 4905 Energy Way
Reno, NV 89502
TELEPHONE NUMBER: (702) 322-6868
MARKETING CONTACT: Bill Delaney

The Sage II and the Sage IV supermicros feature the 16/32-bit MC68000 microprocessor. Memory ranges from 256K to 1MB with storage options from 1-640K floppy to Winchester hard disks up to 40MB. All systems can be configured for multiuser operation with each user having a partition of memory within the main processor. All Sage computer systems run the Idris operating system from Whitesmith, Ltd. and Logos Information Systems.

COMPANY NAME: The Santa Cruz Operation
ADDRESS: 500 Chestnut Street
Santa Cruz, CA 95060
TELEPHONE NUMBER: (408) 425-7222
MARKETING CONTACT: Doreen Hamamura

The Santa Cruz Operation (SCO) features the Xenix operating system and a full range of application software for UNIX™-based systems. SCO's selection includes the Multiplan™ spreadsheet package; Uniplex™, a comprehensive, menu-driven word processor; fully GSA-certified Level II Cobol™; and the Informix™ relational database management system. The company also features the UNIX System Tutorials, an introductory training package for new UNIX system users; an integrated family of cross-assemblers; and other UNIX-based products. For the UNIX-system OEM, SCO provides comprehensive system support and software development services.

COMPANY NAME: Silicon Graphics, Inc.
ADDRESS: 630 Clyde Court
Mt. View, CA 94043
TELEPHONE NUMBER: (415) 960-1980
MARKETING CONTACT: Randy Nickel

Silicon Graphics demonstrated the IRIS (Integrated Raster Imaging System) workstation, with its realtime 2D and 3D graphics capability. The IRIS uses the Geometry Engine™, a proprietary VLSI chip designed by Silicon Graphics, to provide realtime graphics processing. This chip is integrated into a system of innovative hardware, powerful graphics software, the UNIX operating system, and the Ethernet communications network. The IRIS provides a powerful general-purpose computing environment suitable for complex engineering and scientific applications.

COMPANY NAME: SKY Computers, Inc.
ADDRESS: Foot of John Street
Lowell, MA 01852
TELEPHONE NUMBER: (617) 454-6200
MARKETING CONTACT: Howard Klemmer,
Senior V.P.

SKY is in the business of providing state-of-the-art high-speed arithmetic peripherals. The current product line includes a Fast Floating Point Processor (SKYFFP), and a 1M Flop array processor (SKYMNK). The SKYFFP provides high-speed floating point arithmetic to M68000-based systems on four different standard buses as well as some proprietary buses. It is totally transparent to the user and provides many compound functions in microcode for higher speed. The SKYMNK is a board-level array processor for integration into most popular microcomputers including Q-bus, Multibus (Intel and M68000) and Versabus based systems, as well as some proprietary buses. The SKYMNK comes with a very rich set of subroutines for optimal array processor use.

COMPANY NAME: Sof Test, Inc.
ADDRESS: 555 Goffle Road
Ridgewood, NJ 07450
TELEPHONE NUMBER: (201) 447-3901
MARKETING CONTACT: Michael Heffler

Sof Test exhibited its four UNIX-based software packages: SofGram—the only UNIX product to transmit and receive messages over the Telex, TWX and phone networks; LEX word processing system; THE MENU SYSTEM—a user-friendly front-end to any UNIX-based application that allows a designer to build window-based menus in English; and SofForms—a forms creation and management package.

COMPANY NAME: Quadratron Systems, Inc.
ADDRESS: 15760 Ventura Boulevard, #1032
Encino, CA 91436
TELEPHONE NUMBER: (818) 789-8588
MARKETING CONTACT: Vanessa Abbe

Q-Office is an office automation software package that includes Q-One word processing; Q-Date electronic calendar; Q-Menu menu generator; Q-Note electronic notepad; Q-Form form generator, Q-Call phone directory; Q-Math calculator; and Q-Mail electronic mail. Q-Office is completely device independent and comes with termcap and printcap files and generators.

COMPANY NAME: Relational Technology, Inc.
ADDRESS: 2855 Telegraph Avenue
Berkeley, CA 94705
TELEPHONE NUMBER: (415) 845-1700
MARKETING CONTACT: Peter Tierney
Vice President Marketing

Relational Technology, Inc. displayed INGRES, a relational database management and applications development system. INGRES provides flexible report writing capabilities; data management and applications generation through Visual Programming™ forms interfaces, and a powerful query language, QUEL, for interactive use or easy embedding in most programming languages. INGRES includes INGRES/REPORTS™; INGRES/QUERY™; INGRES/FORMS™; INGRES/APPLICATIONS-BY-FORMS™; INGRES/GRAPHICS™; INGRES/NET™ for distributed access; and EQUEL (Embedded QUEL) interfaces for FORTRAN, COBOL, BASIC, PASCAL, and C languages. INGRES runs on VAX computers under the UNIX or VMS operating system, or on MC68000-based supermicrocomputers under UNIX operating systems.

COMPANY NAME: STSC, Inc.
ADDRESS: 2115 E. Jefferson Street
Rockville, MD 20852
TELEPHONE NUMBER: (301) 984-5000
MARKETING CONTACT: Rich Paulson

STSC has developed a high-performance enhanced APL interpreter for the UNIX environment, to complement its very popular APL*PLUS/PC system and its APL products for IBM mainframes. The APL*PLUS/UNIX system runs on a wide variety of supermicro and minicomputers that use UNIX System 3 or System V. This all-new APL system provides features that previously have been available only on mainframe systems—including nested arrays and very large APL variables and workspaces.

COMPANY NAME: Sage Computer
ADDRESS: 4905 Energy Way
Reno, NV 89502
TELEPHONE NUMBER: (702) 322-6868
MARKETING CONTACT: Bill Delaney

The Sage II and the Sage IV supermicros feature the 16/32-bit MC68000 microprocessor. Memory ranges from 256K to 1MB with storage options from 1-640K floppy to Winchester hard disks up to 40MB. All systems can be configured for multiuser operation with each user having a partition of memory within the main processor. All Sage computer systems run the Idris operating system from Whitesmith, Ltd. and Logos Information Systems.

COMPANY NAME: The Santa Cruz Operation
ADDRESS: 500 Chestnut Street
Santa Cruz, CA 95060
TELEPHONE NUMBER: (408) 425-7222
MARKETING CONTACT: Doreen Hamamura

The Santa Cruz Operation (SCO) features the Xenix operating system and a full range of application software for UNIX™-based systems. SCO's selection includes the Multiplan™ spreadsheet package; Uniplex™, a comprehensive, menu-driven word processor; fully GSA-certified Level II Cobol™; and the Informix™ relational database management system. The company also features the UNIX System Tutorials, an introductory training package for new UNIX system users; an integrated family of cross-assemblers; and other UNIX-based products. For the UNIX-system OEM, SCO provides comprehensive system support and software development services.

COMPANY NAME: Silicon Graphics, Inc.
ADDRESS: 630 Clyde Court
Mt. View, CA 94043
TELEPHONE NUMBER: (415) 960-1980
MARKETING CONTACT: Randy Nickel

Silicon Graphics demonstrated the IRIS (Integrated Raster Imaging System) workstation, with its realtime 2D and 3D graphics capability. The IRIS uses the Geometry Engine™, a proprietary VLSI chip designed by Silicon Graphics, to provide realtime graphics processing. This chip is integrated into a system of innovative hardware, powerful graphics software, the UNIX operating system, and the Ethernet communications network. The IRIS provides a powerful general-purpose computing environment suitable for complex engineering and scientific applications.

COMPANY NAME: SKY Computers, Inc.
ADDRESS: Foot of John Street
Lowell, MA 01852
TELEPHONE NUMBER: (617) 454-6200
MARKETING CONTACT: Howard Klemmer,
Senior V.P.

SKY is in the business of providing state-of-the-art high-speed arithmetic peripherals. The current product line includes a Fast Floating Point Processor (SKYFFP), and a 1M Flop array processor (SKYMNK). The SKYFFP provides high-speed floating point arithmetic to M68000-based systems on four different standard buses as well as some proprietary buses. It is totally transparent to the user and provides many compound functions in microcode for higher speed. The SKYMNK is a board-level array processor for integration into most popular microcomputers including Q-bus, Multibus (Intel and M68000) and Versabus based systems, as well as some proprietary buses. The SKYMNK comes with a very rich set of subroutines for optimal array processor use.

COMPANY NAME: Sof Test, Inc.
ADDRESS: 555 Goffle Road
Ridgewood, NJ 07450
TELEPHONE NUMBER: (201) 447-3901
MARKETING CONTACT: Michael Heffler

Sof Test exhibited its four UNIX-based software packages: SofGram—the only UNIX product to transmit and receive messages over the Telex, TWX and phone networks; LEX word processing system; THE MENU SYSTEM—a user-friendly front-end to any UNIX-based application that allows a designer to build window-based menus in English; and SofForms—a forms creation and management package.

COMPANY NAME: Tom Software
ADDRESS: P.O. Box 66596
Seattle, WA 98166
TELEPHONE NUMBER: (206) 246-7022
MARKETING CONTACT: Jerry Hill, VP

Tom Software is a software manufacturer, distributing its products through a worldwide network of licensed consultants. Tom provides applications and utility solutions for the first-time user and data processing professional. The applications include distributors, residential contractors, manufacturers, specialty contractors, general contractors, property management, restaurants, not-for-profit organizations, word processing, and an applications development utility.

COMPANY NAME: Software Express
ADDRESS: 10103 Fondren
Suite 220
Houston, TX 77096
TELEPHONE NUMBER: (713) 270-5218,
(800) 231-0062
MARKETING CONTACT: Steve Thomas

APPGEN Applications Generator—First non-procedural 4th-generation UNIX applications development and maintenance environment that produces applications without coding. Produces transaction-driven, interactive, commercial applications with a limitless number of files, screens, menus and reports. Complete environment includes screen creator, report creator, relational DBMS, parametric definition methodology, automatic end-user documentation, English query, interactive structured design facility, optimized "C" runtime module, import-export facility.

Benefits: Develop 10 times faster, modify 50 times faster.
Operating systems: UNIX (3.0, 4.2, 5.0, 7.0), Xenix, Pick.
Cost: $6,000 development, $600 runtime.
Applications available: G/L, A/P, A/R, Payroll, Inventory and 4 verticals.
Support: Training, maintenance, video tapes, professional documentation.

COMPANY NAME: Spectrix, Inc.
ADDRESS: 3000 Dundee Road
Northbrook, IL 60062
TELEPHONE NUMBER: (312) 291-0850
MARKETING CONTACT: Richard C. Drew

Spectrix Inc. is a designer, manufacturer and marketer of hardware operating systems and software. Our S10 and S30 are high-performance product families based on MC68000 technology, run under UNIX and use Multibus. In addition we have created Basic 3 which is a Dartmouth Basic language that features virtually 100% compatability with Wang Basic 2. Basic 3 gives Spectrix users access to a library of thousands of application software programs that have been proven over the years on Wang.

COMPANY NAME: Sphinx, Ltd.
ADDRESS: 43-53 Moorbridge Road
Maidenhead, Berkshire
SL6 8PL
ENGLAND
TELEPHONE NUMBER: +(44) 628-75343
MARKETING CONTACT: Cornelia Boldyreff

Sphinx is a software marketing company specialising in products related to UNIX. Sphinx was set up in May 1983 to provide marketing and service links between developers of multiuser software and potential professional users. Sphinx acts as an independent supplier qualified to advise users on suitable products for their application needs. Sphinx offers software developers and suppliers professional marketing skills and services. These include specification and testing of software, documentation and packaging, promotional activities and sales. Sphinx guarantees a high level of service throughout its sphere of operations.

COMPANY NAME: Sritek, Inc.
ADDRESS: 10230 Brecksville Road
Cleveland, OH 44141
TELEPHONE NUMBER: (216) 526-9433
MARKETING CONTACT: Walter Fuchs

Sritek manufactures high-performance coprocessors and advanced operating systems for the IBM PC and XT. The Xenix System 3.0 for the Sritek 68000 coprocessor offers paged-memory management which can support up to 16 users. For the 16032 coprocessor we offer the Berkeley 4.1 BSD which supports 32-bit demand-paged virtual memory management and floating-point hardware. We will soon offer the 4.2 BSD version with networking facilities. UNIX System V will be available on the 68000, 16032 and 80286 coprocessors during the second quarter of 1984.

COMPANY NAME: TYX Corporation
ADDRESS: 11250 Roger Bacon Drive
Suite 16
Reston, VA 22090
TELEPHONE NUMBER: (703) 471-0233
MARKETING CONTACT: James Gauthier
Vice President Sales

TXY Corporation, a vendor of typesetting software and typesetting hardware packages, presented two of their systems, the TYXSET 1000 Textsetting System and the TYXSET 100 Personal Textsetting System. The software runs on UNIX System 3, System 5 and Xenix. It includes the TEX typesetting language written in "C", a user-friendly menu package for typesetting composition using TEX, and a powerful text editor for creating and editing documents.

The TYXSET 1000 package is a multiuser system designed especially for agencies with high volume print output. It includes the TYXword/T Software System, the T/10 Laser Printer, and the T/25 Computer. The TYXSET 100 package is a single-user system that includes a VICTOR 9000 personal computer equipped with a 10-megabyte hard disk and a text editor.

COMPANY NAME: Texas Instruments, Inc.
ADDRESS: 17781 Cartwright Road
Irvine, CA 92714
TELEPHONE NUMBER: (714) 660-8243
MARKETING CONTACT: George White

Nu Machine Computer Family—This computer was architected at the Laboratory for Computer Science at M.I.T. and is based on the M.I.T. Nubus. Two versions were announced January 10, 1984 and shown for the first time at UniForum '84. This high-end UNIX-oriented product supports one or two high-resolution bit-mapped displays and is available with an 84 or 474 Mbyte disk. A two-user configuration provides high resolution displays at under $20,000 per user in OEM quantities.

COMPANY NAME: UniComp Corporation
ADDRESS: 202 Plaza Towers
Springfield, MO 65804
TELEPHONE NUMBER: (417) 883-6800
MARKETING CONTACT: Donald Anderson

UniGen is an application generator for the UNIX Operating System. UniGen allows the generation of UNIX applications using menus, extensive user prompting, and maintenance of random access record files.

UniGen includes programs for building menus, user-defined random access files with keyed records, and generating user prompting, all to the user's specifications. Separate runtime modules for installation on customer site. Menu Options may call other menus, run programs, or run prompts. Prompts ask the user for arguments necessary to execute UNIX, and other programs, then cause the programs to execute. An extensive library of prompts and menus is shipped with UniGen. All prompts and menus shipped with UniGen can be changed to meet user specifications or completely new and different prompts may be generated. Custom menus may be created using any editor or word processor that creates standard UNIX text files. The user may define keyed access files and may specify the number of records in the file, prompts for each field, and each field's size. Records may be accessed randomly by key field and maintained with add, change, view, and delete functions.

COMPANY NAME: UNIFY Corporation
ADDRESS: 9570 SW Barbur Boulevard
Suite 303
Portland, OR 97219
TELEPHONE NUMBER: (503) 245-6585
MARKETING CONTACT: Mr. Duke Castle,
Director of Marketing

UNIFY is a multiuser, menu-driven and fully interactive UNIX relational database management system. It is designed to support large databases up to 2 billion records over 8 physical devices.

UNIFY provides a complete range of application development tools including a menu handler, automatic data entry, query by forms, report generator, and structured query language (SQL) based on IBM's SEQUEL 2 relational query language. A powerful host language interface, compatible with C and RM-Cobol, is also provided.

UNIFY offers multiple data access and raw file I/O capability for fast performance while only requiring a minimum memory configuration of 256K bytes and 2 megabytes of disk space.

COMPANY NAME: UniPress Software, Inc.
ADDRESS: 1164 Raritan Avenue
Highland Park, NJ 08904
TELEPHONE NUMBER: (201) 985-8000
MARKETING CONTACT: Joyce Bielen

UniPress Software, Inc. exhibited its UniPlus+ UNIX System V operating system for the Apple Lisa computer, as well as other UNIX software available for the VAX and many 68000 implementations. Products on display were EMACs, a full screen multi-window editor with built-in MLISP programming language, LEX word processing system, a menu-driven interactive word processor with full screen editing and scrolling, the highly regarded Lattice C compiler, both native and cross for UNIX and MS-DOS environments. Other products on display included Phact ISAM, /ROB, UniCalc spreadsheet, MIMIX CPM emulator, and the menu system.

UniPress Software is a software publishing firm for UNIX, VMS, and MS-DOS environments. Our UNIX software is available on MC68000 implementations including the Sun, Masscomp, Plexus, Dual, Tandy 16, Apple Lisa, Apollo, etc. as well as the VAX computer.

COMPANY NAME: Uniq Computer Corporation
ADDRESS: 28 South Water Street
Batavia, IL 60510
TELEPHONE NUMBER: (312) 879-1008
MARKETING CONTACT: Roger A. Knuth

Uniq Computer Corporation announced UNIQORN, a packaged system based on the long-awaited Digital LSI-11/73, the single-board implementation of the PDP-11/70 CPU. UNIQORN offers PDP-11/44 class performance at a much lower price. In addition, Uniq provides UNIX System V, System III and Berkeley implementations on Digital VAX and PDP-11 computers. A variety of software products are available, including UNIFY DBMS, UNICALC virtual memory spreadsheet, RM/Cobol, BASIC, Pascal, Sibol, word processing and networking. Uniq provides ongoing support for all of its products as well as consulting and training in the UNIX environment. Training is held both in scheduled classes and on-site at client locations.

COMPANY NAME: Unisource Software
Corporation
ADDRESS: 71 Bent Street
Cambridge, MA 02141
TELEPHONE NUMBER: (617) 491-1264
MARKETING CONTACT: Dena Wortzel

Unisource Software Corporation is a major publisher and distributor of UNIX software for the corporate, professional and system development markets. The company's UniForum exhibit featured its Office UNIX System, a package consisting of a variety of business applications running under VENIX/86, a licensed implementation of AT&T's UNIX operating system. Single and multiuser configurations of IBM PCs, Compaqs and display terminals were shown. Fusion Ethernet software newly ported to the IBM PC was shown for the first time, linking Unisource's system with those of several other exhibitors. Among the UNIX-based applications available from Unisource are The FinalWord word processor; Leverage for database management; Viewcomp, an electronic spreadsheet; Sunburst accounting packages; and the company's own Office Menu Tool designed to make UNIX easy to use for inexperienced users.

COMPANY NAME: UniWare
(Division of Nuvatec, Inc.)
ADDRESS: 261 South Eisenhower
Lombard, IL 60148
TELEPHONE NUMBER: (312) 620-4830
MARKETING CONTACT: Eric Thiele

UniWare offers UNIX-based microprocessor software cross-development tools. Approximately ten popular UNIX-based hosts and twenty target microprocessors are currently supported; support for others is announced regularly. A cross-development package consists of a macro preprocessor, cross-assembler, link editor, downloaders, "generic" downloader source code, and various utility programs. 'C' cross-compilers are available for selected target processors.

Significant new product announcements will be made later this year.

COMPANY NAME: Vandata
ADDRESS: 17544 Miovale Avenue, North
Suite 107
Seattle, WA 98133
TELEPHONE NUMBER: (206) 542-7611
MARKETING CONTACT: Dwight Vandenberghe

Vandata is a software firm that specializes in computer languages, both as commercial products and as a contract service to industry. Since 1975, Vandata has been responsible for the development of a number of interpreters and compilers, as well as two parser generators, a C subset for small ROM-based systems and a Whitesmiths-based Z80 C Cross-compiler.

The Vandata Z80 C Cross Compiler is a full implementation of the C language, targeted for the Z80 microprocessor. The compiler can be run on PDP-11, VAX, Z8000 and 68K hosts, as well as on CP/M systems. Based on Whitesmiths compiler, Vandata C offers thorough support for ROM/RAM environments, extremely compact generated code, and the ability to intermix assembly language routines with C routines. An optimizer, Z80 assembler, linker librarian, and other tools are provided with the compiler, as well as the facility to interface to other cross-assemblers. A royalty-free routine library is available.

COMPANY NAME: Voelker Lehman Systems, Inc.
ADDRESS: 44000 Old Warm Springs Boulevard
Fremont, CA 94538
TELEPHONE NUMBER: (415) 490-3555
MARKETING CONTACT: Joe L. Voelker

Voelker-Lehman is a hardware/software systems integrator with a variety of products and services including:

- VAX-based multivendor hardware systems
- Hardware configuration, integration, installation and support
- TURBO-VX, performance accelerator for VAX 11/750
- U/OS, enhanced UNIX operating system software compatible with Berkeley 4.1, featuring System V enhancements and U/INSTAL, automatic installation utility
- U/CDB, interactive source-level C language debugger
- Vanilla 4.1 and 4.2 BSD
- Full software support

COMPANY NAME: The Wollongong Group
ADDRESS: 1129 San Antonio Road
Palo Alto, CA 94303
TELEPHONE NUMBER: (415) 962-9224
MARKETING CONTACT: David J. Preston,
Director of Marketing

EUNICE furnishes VAX users with the ability to merge the VMS environment with the advantages of UNIX commands and utilities. EUNICE permits users to transport previously written software from either the VMS or UNIX domain, without affecting the software. EUNICE allows any mixture of VMS and UNIX users to share the resources of a single computer system.

PEGASUS offloads context switching tasks associated with terminal I/O communications in a keyboard transparent environment. PEGASUS offloads either VI or Wollongong's Full Screen Editor for host systems running VMS, UNIX, or EUNICE.

Wollongong's other system-level software products include: RM/COBOL, MISTRESS, HORIZON, Full Screen Editor, and IP/TCP. Wollongong markets consulting and engineering services to other UNIX companies, and major hardware manufacturers desiring to participate in the UNIX marketplace. These services include performance analysis, custom hardware/software integration, and marketing assistance with new UNIX products. Educational courses and seminars are available for new and experienced UNIX users.

COMPANY NAME: Yates Ventures
ADDRESS: 4962 El Camino Real
Suite 111
Los Altos, CA 94022
TELEPHONE NUMBER: (415) 964-0130
MARKETING CONTACT: Pamela Pasotti

Yates Ventures conducts primary market research and performs hands-on product evaluations of UNIX hardware and software. Multiclient subscribers receive definitive market data and analysis as an aid in strategic decision making. The Yates Perspective, a monthly newsletter, keeps clients up to date on events in the UNIX market. The UNIX Encyclopedia is a popular end-user source of UNIX product information.

The Excelan TCP/IP Protocol Package

Bill Northlich
Northlich CSS, Inc.
21 Newell Court
Walnut Creek, CA 94595

Bruce Borden
Silicon Graphics, Inc.
630 Clyde Court
Mountain View, CA 94043

The Excelan
TCP/IP
Protocol Package

Bill Northlich, Northlich CSS Inc.
21 Newell Court, Walnut Creek, CA 94595

Bruce Borden, Silicon Graphics, Inc.
630 Clyde Court, Mountain View, CA 94043

ABSTRACT

The Berkeley/BBN IP/TCP internetwork protocol code was ported to an intelligent front-end Ethernet board. The EXOS/101 Ethernet Front-End Processor from Excelan is a Multibus DMA board with an 8MHz 8088 and 128KBytes of ram memory. By porting the protocol code onto the front-end, the host overhead was reduced drastically, and the kernel-resident protocol software was reduced to a small device driver. The implementation mimics the Berkeley socket interface utilizing pseudo-devices and ioctl calls, thus allowing most of the existing network user code to run without modification.

The prom-resident operating system on the EXOS/101 board provides support for multiple processes and communication via message queues. The initial implementation used a separate process per TCP connection, which had to be rewritten to properly handle multiple forks reading and writing the same connection. The current implementation maintains one process which knows how to save its state when "sleep" is called, and to resume a blocked incarnation when a "wakeup" is issued. Both of these approaches were possible without any changes to the Berkeley code, only to the interface and support routines.

The host to front-end protocol was designed to place most of the burden on the front-end, and to be host operating system independent.

This paper describes the port to the front-end, and examines the desirability of this approach for future network protocol implementations.

Terminology is Important

In the last several years with the advent of the ARPANET and interconnection schemes such as Ethernet there has been a lot of research and development which has solidified and made substantial many ideas in computer networking. Many concepts which emerge forthright and absolute from research tend to become reduced to buzzwordology by popularization and the trade press. Terms which come to mind in this guise are "virtual", "distributed", and the term "networking" itself. By "networking" we mean resource-sharing networks, ala the ARPANET, and not merely the hooking-up of device A to device B.

The term "protocol" should be devoid of confusion these days, at least when talking about computers, and not, say, about what to do when the King of Siam arrives. That is, a computer communications protocol is a set of rules, normally implemented in software, which can be vaguely tagged with one or more of the infamous seven layers of the ISO reference model. However, one of the authors recently had a competent engineer in a company whose main business is local networking refer me to "the RS-232 protocol". One continues to read articles and advertisements in respected publications such as "Data Communications" which refer to what is basically a timesharing system with terminals attached as a local network. It does us no good to say we have some networking protocols implemented on a computer front-end if people think it is just another RS-232 board.

History

Among the enhancements to existing systems considered by most developers is enabling their systems to participate in some form of resource-sharing computer network.[PA1] Many technical people in industry are familiar with the Defense Advanced Research Projects Agency's (DARPA) "ARPANET", the very successful prototypical (and archetypical) packet-switching network.[CE1] Ethernet[ME1] is commercially available and provides an inexpensive in-building baseband communication medium almost exactly analogous, logically, to the communications sub-network of the ARPANET.

In (1979) the "Father of the Ethernet", Dr. Bob Metcalf, had started a company called 3com Corporation which offered not only Ethernet hardware, but also a networking software package for UNIX† called "UNET". UNET implemented the standard ARPA network and transport layer protocols (IP/TCP)[AR1] over whatever hardware customer wanted to build, but clearly designed to be run over Ethernet. The senior author, Borden, and Greg Shaw of 3com had implemented "from scratch" a complete protocol suite including the basic transport protocols, file transfer, remote login, and mail applications in just a few months. For almost two years, UNET was the most complete, compatible, and generally useful networking package available (in the first authors' opinion); certainly it was for UNIX.

One disadvantage of UNET is that the main transport protocol (TCP) runs as a user process under UNIX, since it is too big to fit into the small address space of machines like the PDP11. This means that in order to get out to the actual network, a byte has to go from an application program, down to the UNIX kernel, back up to TCP, back down to the kernel, and out to the net. The main problem here is not so much the copying of bytes around, which is bad enough, but the extra context-switch that takes place when TCP runs.

During the last several years, various other implementations of IP/TCP have been under way. Rob Gurwitz of BBN Laboratories in Cambridge, MA did an implementation of IP/TCP for VAX-UNIX*. Bill Joy at the Computer Science Research Group at Berkeley took the Gurwitz code and enhanced it for speed and efficiency in the context of the new UNIX kernel developed at Berkeley. Sam Leffler took over from Joy at Berkeley and made major improvements to the code. Bill Croft at Stanford Research Institute obtained the code and transported it to the PDP-11 from the VAX. Croft's main contribution was getting the basic code to run in the small address space of the PDP-11; as an ancillary task he had to somewhat "de-Berkeley" the way the network code hooked into the standard UNIX system. This work proved very useful to us.

At this point the authors took the Croft code and ported it to the Excelan front end processor. Now a system which wishes to run TCP need not do extensive Kernel modifications or have large support processes running in the background. It is only necessary to compile the small (about 5-6k) device driver and plug in the board.

† UNIX is a trademark of Bell Laboratories.

* VAX is a trademark of Digital Equipment

Why IP/TCP?

The answer: because the ARPA internet protocols (IP/TCP together with FTP, TELNET, SMTP and others) are the only set of protocols extant which:

- Are fully specified.
- Are complete.
- Are actually implemented on a wide range of computer architectures and operating systems.
- Actually all work, ie, interoperate among the different systems mentioned in (3).

That is, the ARPA internet protocols are the only truly standard computer networking standards available today.[PA2]

It is not the purpose here to take each possible candidate for "best protocol" and say what's wrong with it, but we might give one illustration of what we mean by non-standard "standards". Take for example the Xerox XNS series of protocols[XE1]. They are well-specified and widely implemented. They are normally efficient and code-wise compact. Unfortunately, they don't address points 2 and 4 above. That is, first, they are not complete because Xerox has only (publically) specified protocols through the so-called transport layer. No file transfer protocol has been specified, for example. As a result, even though there are many XNS implementations around, many even for UNIX, all have had to beg, borrow, or design their own versions of standard programs such as remote login and remote file transfer. Thus, point (4) is violated, namely, most of the XNS implementations don't (or can't) say anything meaningful to each other.

We like XNS. As was said above, XNS implementations are usually small and fast. We suggest, even, a reading of the XNS specifications as an excellent introduction to the field of protocols and networking. The problem is that there is no standard against which to measure whether or not a particular implementation works, ie, there is no Berkeley Unix for XNS. There is Bridge, if one wants to see whether he can send bits between machines, but how does one transfer files?

We hope one of the companies working in the XNS area will propose some standards and that they will achieve sufficient backing to actually become standards. We hope Xerox itself will give XNS the backing it deserves. However the fact is at this point even if one has a fully Bridge-certified XNS, one can't talk to arbitrary others. Also, we ought to mention the reverse prestige syndrome, ie, some companies have refused to get their XNS certified by Bridge since Bridge represents the competition they are trying to beat.

TCP is large and somewhat unwieldy. On the other hand, after having passed through the hands of Joy and Co. at Berkeley it's relatively fast, though most XNS's are faster. But look what TCP talks to:

- ALL big machines on the ARPANET have a TCP (IBM, DEC-10, UNIVAC, etc)
- UNET (z8000, pdp11, VAX, 68000, 8086)
- Imagen printers (!)
- IBM pc
- 4.1bsd; 4.2bsd
- Excelan-equipped boxes

IP/TCP is in some sense more general in that IP does more routing than XNS DGP and TCP is more useful over long-haul networks. XNS tacitly assumes Ethernet. And lest we forget, TCP is supported by the DOD. We predict that if Xerox does not start supporting XNS by the end of the year, TCP will take over the world of LAN's, as it is already beginning to take over the world of long-haul networking, at least in the US.

Proponents of various protocol suites may argue that their protocols address all the points above, but no one can argue that the ARPA IP/TCP suite does NOT fully meet, and even exemplify, what a standard should be.

Implementation

Networking is implemented in Berkeley Unix in, to use an over-used term, a "layered" manner. There are three layers:

- System Interface
- Protocol Layer
- Interface (hardware driver) Layer

The goal of the System Interface is to provide a consistent operating system interface to user-level programs which wish to use the network (regardless of what goes on in the lower layers.)

There are several system calls which provide this interface. They include (in Berkeley release 4.1a)

socket()	create an end point for communication.
connect()	connect to a remote peer process.
accept()	accept a connection on a virtual circuit from a peer.
send()	send data.
receive()	receive data
sockaddr()	what internet address does this socket correspond to?
select()	tell when something interesting (input or output) happens on a socket.

The name "socket" for the socket() system call is perhaps an unfortunate choice since the TCP literature speaks about TCP "sockets" which are what the Berkeley scheme refers to as "ports". In the Excelan implementation, a socket() system call is really a library routine which does an ioctl on the socket administration device and returns a file descriptor which may be used by the standard read() and write() system calls to get data from and write data to the network.

A TCP port ("TCP socket") on the other hand is part of the semantics of a TCP connection. If a host on the network is thought of as an apartment building, then the TCP ports at that host are people's individual mailboxes at the apartment building. Thought of another way, process A in host Q corresponds over port X to talk to process B in host R speaking through port Y.

The Excelan package supports the DOD Internet standard User Datagram Protocol (UDP), with which the library routines send() and receive() are usually used. TCP sockets, being Virtual Circuits or byte-streams, are normally used with read() and write(). Standard UNIX operating system calls which apply to sockets are

read()	read data
write()	write data
close()	close a socket
ioctl()	set special options.

The Protocol Layer is designed to hook in beneath the system interface layer transparently, though putting a new protocol in place is certainly a non-trivial task. Towards this Ideal, all protocol-specific routines are called through an indirect table called, strangely enough, the "protocol switch". For instance, there has been effort by various parties to put protocols from Decnet, Xerox XNS, and Symbolics/MIT's Chaosnet into the Berkeley framework. However, none of these implementations are generally available, to the author's knowledge.

The protocol layer is connected to the outside world and the actual hardware via the Interface Layer. The Interface Layer is really just a hardware driver in the classic sense, combined with a clearly specified software interface to the rest of the system.

Remote Logins

For remote logins, one problem is how to make normal programs on the host think that they are talking to a regular (rs-232) type terminal port when in fact they are talking to a remote program over the network. The solution we follow is to use a software device which has been used in various operating systems for about 10 years, called a "pseudo-teletype". A pseudo-teletype driver implements pairs of UNIX devices which do what is needed, ie, it presents a standard tty interface for standard programs like editors, but the bytes to and from the standard programs arrive at another device called the controlling half of the pseudo tty. Another program can then be written to manipulate the original programs by sending and receiving bytes to/from them as if the second program were an actual hardware device.

When used in a networking environment, pseudo-teletypes allow a daemon program to read and write from the network and send/receive the result to/from the normal tty interface. Eventually the remote login and telnet daemons will be moved onto the board, the pseudo-tty driver will go away, and bytes comming in from the net will "look" for all intents to the host like a regular tty.

Addressing

As networks have grown larger (more hosts), bigger (more distributed), and more complex (sub-networks and super-networks), addressing has become an ever increasing problem. The Xerox publication on their Clearinghouse "name" server provides an excellent (if long winded) discussion of name mapping alternatives. Specifically, given a collection of connected hosts, how can I find each host's name, and how do I convert that name into an address or identifier useful to my network software and hardware?

On most networks, the link level hardware relies on a binary address to route messages from source node to destination node (or to set up a virtual circuit on a circuit switched network). The question is: how is that address assigned, changed, and, most, important, obtained? If host "split" has Ethernet address 012345, how does another host discover this information? If host "split" changes its Ethernet board, thus changing its Ethernet address, how is this information distributed to the rest of the interconnected hosts and networks?

In the beginning, IP/TCP had little or no "naming" facilities. Each host had a table of all known hosts on all interconnected networks. Simple routing was performed by inference from the network numbers associated with each host. More recently, a new protocol (ICMP, actually an addition to IP) was designed to exchange host status and localized routing information between IP nodes.

This helps with routing, but ignores the host naming problem. Under IP/TCP, there is still no standard mechanism to turn the name of a host into its internet address. As the number of hosts connected to the world internet increases rapidly, it is patently impossible for each and every host to maintain a complete name to address translation table. This would be like requiring every phone user to maintain a complete list of all telephone users and their phone numbers. What's missing is equivalent to the telephone directory service.

Another related problem is the use of IP/TCP on nodes too small to maintain any name database, even their own name and internet number. Imagine a set of terminals connected to an Ethernet which boot the same identical software via Ethernet from one host. Once the software is running in the terminal, how does it find out who "I" am? What is my name? What is my internet number? It is possible to compile (yes, compile) in a name database to such software, but this becomes a messy management problem when new nodes are added, or network interfaces change. It is also possible for the freshly loaded code to look at its Ethernet address (or some other unique system identifier if the manufacturer was kind enough to supply one) and ask a server on the network to provide a name and Ethernet address based on this unique ID.

This latter scheme in fact is the only "clean" solution, and lacks only standardization to fit well into the scheme of things. Provoked by just such a networked terminal application, an RFC is in progress to define such a "name" server.

A related need is for a simple boot protocol which the terminal can use to obtain its initial program. This protocol should fit cleanly into the IP scheme of things making it easy for IP hosts to act as boot servers It must also be so simple that it can be implemented in a few K of prom (remember, the prom doesn't know what it's internet address is!).

Last addressing/naming problem. Assuming that each host knows its internet address, and that it knows or can find the name and internet address of any host it desires to communicate with, how is this translated into actual communication on any given physical network. That is, how is the internet address for a host (source or destination) converted into the appropriate network address for the physical network hardware in use.

For example, consider the case of Ethernet. An internet address is 32 bits long, and an Ethernet address is 48 bits long. Once I've decided that the destination I wish to address is on the Ethernet, how is its internet address converted to an Ethernet address? Under the original Berkeley (BBN?) code, the internet address was converted to the Ethernet address via a simple mapping function. The low three bytes of the internet address were used as the low three bytes of the Ethernet address, and the high three bytes of the Ethernet address were assumed to be fixed. This was extended (a little) by taking one bit of the low three bytes to select between two possible fixed three high byte values. With the recent proliferation of new vendor Ethernet hardware, this scheme became a burden.

A new protocol to the rescue. The Address Resolution Protocol (ARP) was defined to alleviate this problem. Under ARP, a low-level Ethernet driver maps an internet address to an Ethernet address via a simple broadcast/echo protocol and a small mapping cache in the driver. That is, when an internet address is handed to the driver, it is looked up in a mapping cache. If a mapping to an Ethernet address is not found, a "whereis" packet is broadcast asking everyone if they know what Ethernet address corresponds to the internet address in question. Typically, the host owning the requested internet address responds with its Ethernet address (although a mapping server may respond on behalf of nodes which do not support ARP), and the association from internet address to Ethernet address is added to the mapping cache. Finally, the original packet can be transmitted to the appropriate destination. Once the correspondence is entered into the mapping cache, it will remain valid as long as it is referenced, thus incurring the extra protocol exchange only rarely.

For hosts which do not support ARP yet, there are two interim solutions, the above mentioned name server node, and a mechanism to force entries into the mapping cache of the source host so that ARP will not be used.

On Speed

A short comment about speed of networks: There are applications where a very high bandwidth network is needed, ie, for graphics applications and/or remote file systems. However for most applications, ie, file transfer, mail, remote login, the bandwidth is not critical. The ARPANET has muddled along fine with a bandwidth of 56 kbits/sec and an average during the day of 15-25 kbits/sec. Most of the things we CURRENTLY do on networks involve processing, forking processes, and relatively little data throughput. Of course when we do have data to go out or come in, we want it fast, but the fact is that the ancillary functions take up most of the elapsed time in a normal resource sharing network.

Short Overview of Excelan Board

The following is a list of some of the noteworthy features of the Excelan front-end processor:

- Has an 8088 cpu and 128k of memory on-board for running downloaded protocol software, eg, IP/TCP

- Provides for DMA movement of data to/from host
- Implements Message-based communication with host
- Has an integral Ethernet controller
- Currently implemented for Multibus and VME bus
- Allows very flexible setup with many useful modes
- Has on-board firmware operating system which provides primitives to
  - create, delete, start, and stop processes
  - transfer messages to and from the host, other processes on the board, and the Ethernet
  - DMA data to and from the host
  - get and set the time

Setting the board up can be complicated but virtually every option one might reasonably want to specify is available. One can specify, for the host, polling or interrupt mode, for example, and if interrupts are used, whether the interrupt should be caused by raising the bus interrupt line or writing to a memory or I/O location, and may be individually configured for incoming and outgoing messages. If this were done with jumpers, the board would be mostly jumpers.

One of the most interesting parts of the setup is that the host (may) specify that the board do data conversion on byte streams, shorts, and long words. That is, the host writes a byte stream, some shorts and a long into the configuration message in its "native" language, and the board figures out the vagaries of byte and short ordering that the host is doing (with respect to the board) and provides a firmware primitive to perform the appropriate conversions. Also, if the host stores byte streams onto the bus "byte swapped", as does the SUN cpu board, the board automatically byte-swaps any data it moves to or from the host via DMA.

The model we use is to assume we are making a "procedure call" to the board. What happens is:

1. The host process configures a message.
2. The host process wakes up the board by writing to a known IO port.
3. The board firmware transfers the message to the board.
4. The on-board code does the specified request.
5. Perhaps at some later time, the on-board code sends a reply message to the host for this message.
6. The host process continues.

The Actual Implementation.

What follows is a discussion of the actual code that was written or ported to the Excelan IP/TCP package. The implementation neatly broke into two major pieces, the Host Side and the Board Side.

Briefly described, the host side consists of the driver itself, the socket library, and the user programs.

The user programs which run as of this writing consist of rlogin, rcp and rsh from Berkeley, along with their associated daemons. The major changes to these programs from the original Berkeley versions had primarily to do with the fact that we did not implement the select() call from 4.1a, at least for socket and pseudo-tty file descriptors, in the first release. We circumvented the always-blocking IO in UNIX by using several processes for input, output, and errors. This was a mistake; having select() and some of the socket ioctls in the first release would have made life much easier.

The socket library is a set of routines which simulate the network system calls of 4.1a BSD by ioctls to socket devices in the Excelan driver, and provide certain other miscellaneous services. The calls supported in the first release are

socket() create an end-point for communication; return a file descriptor for a socket device upon which to do further socket library actions.

accept() Accept a connection from a peer process.

connect() Connect to a peer process

send() send data. In TCP a write() on a socket file descriptor will also send data.

receive() receive data. In TCP a read() on a socket file descriptor will also get data from a peer.

sockaddr() given a socket, return it's internet address.

rhost() return an internet address given a symbolic name.

raddr() inverse of rhost()

htons() "host-to-network-short". Do the right thing in so far as byte swapping a short is concerned.

The driver is actually three drivers (five, if one counts the pseudo-ttys):

Administrative device

This device administers socket devices. Rather than looking around for an unused socket device, a user program (actually the socket library routine "socket") just opens the admin device. The admin device also implements the board interrupt routine.

Download device

Used for downloading. Makes the download code very simple.

Socket device

a character device which is used for network activity proper. The result of a socket() library call is a socket device allocated in the host and a corresponding socket created on the board.

The host software has been brought up on the following matrix of systems and processors: on the Motorola 68000: Version 7, System III, System V, and two one of a kind UNIX versions; on the Intel 8086: Version 7. We believe that the entire package is one of the most portable software packages available anywhere,

There were two versions of the on-board code done, the first utilizing most of the process management features of the NX-101 operating system firmware, and the second consisting of mostly one big "hunk" of code which implemented primitive processes and had a primitive scheduler.

On-board processes of some kind were needed since the plan was to transport, as painlessly as possible, the UCB code to the board, and the UCB code for the most part "thinks" it is inside a UNIX kernel running as part of a user process's system call. All the intricacies of UNIX processes did not have to be simulated, but sleep() and wakeup() clearly had to be dealt with, along with such miscellany as iomove(), spl(), and timeouts.

Once a simulation of the inside of UNIX was in hand, we had to write code which dealt with the host and essentially transformed the message passing interface provided by the NX-101 into what would seem to the UCB code proper like just another system call from a UNIX user process. The attempt of course was to keep as much of the UCB code as possible "intact"; mostly knowledge about the NX-101 was restricted to just the original system call routines. The create-a-socket code required some non-trivial work due to sockets on the board and on the host being different things (a socket in the former case; a device in the latter).

Byte-swapping should be mentioned, if only to make it clear that we make no claim to somehow have synthesized a theory which will handle network byte-swapping for all machines. We do have it down to a fairly automatic process -- one must set by hand two variables. A through discussion of byte-swapping would take, if not another entire paper, certainly it's own appendix. For the flavor of it, consider the problem with the Excelan board. There is

With respect to the bus:
- The host native byte ordering in a short
- The host native byte ordering in a byte stream
- The board native byte ordering in a short
- The board native byte ordering in a byte stream

Also:
- The Network "canonical" byte ordering
- The fact that the board may byte-swap a byte-stream on the way into or out of the board.
- The fact that the UCB code has embedded in it an idea that its got to swap bytes (maybe) to get them into network order.

It is worth mentioning that we were the first user software to run on the Excelan board and make serious use of the on-board operating system that Excelan provides. The on-board OS was surprisingly free of actual code bugs. We expected to have to do much more debugging of the Excelan rom code than actually took place. We did find three bugs in the rom software each of which Excelan fixed within one day after having the bug reported.

A major shortcoming of our implementation is that we have not figured out what to do about N boards in a single host, or, "internetworking". The IP protocol, which naturally does internetworking between hardware networks connected directly to it, is on the board. There is only one "network" (Ethernet) connected to the board. One must come up with an implementation which allows IP to optionally have another "network" connected to it which actually sends IP packets up to the host and down to some other board's IP, hopefully without implementing IP itself on the host.

Subtleties of the Implementation

At the beginning of the project the team was rife with enthusiasm and the task seemed straightforward. Old hands all, we set out to do a smooth, and timely implementation of a worthy project, paying somewhat less attention than we probably should have to the voices of experience and more than we should have to the voice of the Muse as he sang of a networked world. That is to say, there were some unanticipated problems. An example of one is the problem of indirect buffer management.

Above, it was said the the basic model for the host implementation is that a user program wishing to access the network should set into motion a chain of events which effectively makes a "procedure call" to code running on the board, and possibly gets back some results therefrom. On a first reading of the Excelan Manual, one finds out about the message queues to and from the board which are managed by the NX-101 firmware. An obvious way to send a message effecting a procedure call to the board might be to

- Find an available outbound message queue entry.
- Fill it with data, including a pointer to the queue entry itself, and send the entry to the board.
- Sleep on it waiting for a reply. The reply will come to the interrupt routine which presumably knows who to wake up from the pointer in the returned message.

The problem with this approach is that the NX-101 won't allow the host to "hold on" to any of it's queue entries. One must define an intermediate queue, at least in UNIX, whose entries

- A process can use to sleep on, waiting for the return message from the board, and
- Can be buffers which the interrupt routine can use to pass back to the user process any "result" parameters of the original procedure call.

So, what actually happens in the current implementation is that the host

- Finds an outbound queue entry
- Finds an intermediate queue entry
- Fills outbound message with data including a pointer to the intermediate queue entry
- Sleeps on the intermediate queue entry
- In the interrupt routine, when a message from the board comprising a reply the original message from the host has come in, the host wakes up processes on the pointer to the intermediate queue entry, and also puts any results into the intermediate queue entry.

The code for implementing the intermediate queue makes understanding the driver somewhat difficult. Hopefully this complication will be removed in future releases of the product.

DMA

The advantage of allowing the board to DMA into it's memory from the host is of course to be able to move data directly from a user process out to the network. One problem with that plan is that some hardware configurations do not allow access, from the bus that the board is connected to, to some or all of the user processes running. Obvious and ubiquitous examples are various incarnations of the SUN multibus cpu board. Another problem is that data movement out of the host is at the convenience of the board.

Any implementation of a UNIX interface which is general enough to be able to fit into a wide variety of systems will require one data copy by the host. If the host and the board were able to cooperate in managing a data buffer pool, as in some kind of memory-mapped implementation, our conjecture is that speed would be substantially increased and complexity decreased. Of course, most hardware systems understand DMA, but not all work well with memory-mapping (eg, Unibus).

The fact remains, though, that DMA directly to and from a user process offers the best potential throughput with the least potential load on the host.

Plans for the Future

The following are some of the things we hope to do in the near term with the Excelan package, given roughly in the order in which we plan to do them.

1. Port the rest of the extant Berkeley Network user code, ie, ftp, telnet, rwho, rwhod, ruptime.
2. Implement various statistics programs. The Berkeley Network Status Programs are envisioned; in addition, the board's firmware collects fine-grained statistics about the Ethernet which we will print out. One interesting and useful feature of the board is that it has a TDR function which allows determination of the position of cable problems by triangulation.
3. Implement select() and the full Berkeley 4.2 system interface, including listen() and stackable connects.
4. A major milestone will be moving the telnet and rlogin daemons from the host to the board. This will allow essentially the "firmware" implementation of remote logins. The current pseudo-tty driver on the host will be replaced by a terminal-multiplexor driver of the classic variety. A connection established from anywhere on the network to the rlogin daemon (on the board) will essentially cause the "carrier detect" bit in the host driver to go on.

Conclusions

- The implementation took longer than we planned (what's new?).

- The throughput of the board, although disappointing, has proven adequate for most uses.
- The host driver can be (and is) small and very portable and requires no system modifications to install.
- The protocol processor cpu must have as much or more horsepower than the host processor.
- The idea and design are good, and will be much copied in the future.

CSNET Grows Up

Michael T. O'Brien
The Rand Corporation
1700 Main Street
Santa Monica, CA 90406

Daniël B. Long
Bolt Beranek and Newman Inc.
10 Moulton Street
Cambridge, MA 02238

CSNET GROWS UP

Michael T. O'Brien
The Rand Corporation
1700 Main St.
Santa Monica, CA 90406
Net: obrien@rand-unix

Daniel B. Long
Bolt Beranek and Newman Inc.
10 Moulton St.
Cambridge, MA 02238
Net: long@bbn-unix

The Computer Science Research network (CSNET) is undergoing a transition from a development phase to a member-supported utility. In this paper, we describe the many changes that will result from this transition.

New Network Software. CSNET has developed a new communications facility called "X25Net". This software/hardware facility allows a site to connect to the Telenet public packet-switch network, and to use it to communicate with other hosts on both Arpanet and Telenet using the DOD IP/TCP protocols encapsulated in X.25 packets. Several CSNET members are in the process of becoming X25Net sites using the CSNET-developed IP-to-X.25 interface. This development gives member sites who use it the ability to treat Telenet as a type of public Arpanet.

The implementation requires each X25Net host to maintain a table which maps Telenet addresses (which look something like international telephone numbers) to Internet addresses. Telenet is Internet network number 14, and each new X25Net host is assigned an Internet address on net 14. A new address mapping table with the new host's Internet and Telenet addresses is then distributed to all X25Net hosts. An X25Net host must be in this address mapping table before any other X25Net host can communicate with it via TCP/IP. There are two complementary facilities included in the CSNET distribution in addition to TCP/IP support. A user-level "pad" program allows allows the host to communicate with any other Telenet host via the X.29 terminal protocol regardless of whether the other host supports TCP/IP. An X.29 server accepts connections from any of Telenet's over 500 "pad"s and from other X25Net hosts. This X.29 protocol is only good for Telnet applications, however; Telenet does not support an FTP protocol of its own. X25Net hosts can, of course, use the full range of Internet services between themselves and other Internet hosts.

In the original design, one X.25 virtual circuit was to be used for each TCP connection. During the implementation, however, we found that Telenet allows only two unacknowledged packets per virtual circuit. A packet acknowledgement must be received from the foreign host before any more packets may be queued for the circuit. This lead to unacceptable network delays, so a strategy has been implemented to open multiple virtual circuits per TCP connection, and use them in parallel, to increase bandwidth.

Communication with Arpanet hosts is via the BBN VAN gateway. Overall bandwidth is somewhat slower than Arpanet bandwidth, due both to virtual circuit setup time, and to the lower bandwidth of the dedicated phone lines which connect the X25Net hosts to the network.

These lines are either 4800 or 9600 baud. While instantaneous bandwidth is lower than is seen on the Arpanet, overall performance is quite acceptable.

The TCP/IP-X.25 interface software is implemented as a device driver under the BBN TCP/IP implementation known as SYS.10. It was developed at Purdue University and is available to CSNET members. Because it is a device driver, it drops into existing systems very easily. It comes with monitoring software to allow examination of the internal state of the address mapping table, the allocation of virtual circuits, etc.

Discussion of costs for this service can be tricky. At first glance, they seem high. Certainly, X25Net service is not for the casual user, as a 4800 baud Data Access Facility to Telenet currently costs $925/month plus packet charges. However, PhoneNet telephone charges amounting to about 20–30 Mbyte/month transmitted over Telenet would run only about $275 in packet charges. Adding in the Data Access Facility, the total cost would be $1200, still $300 less than PhoneNet. X25Net provides the advantage of immediate mail delivery and much lower incremental traffic cost. In addition, FTP, Telnet, and all the other Internet services are available.

Let us contrast this with the Arpanet. While the Arpanet appears free, the actual cost is about $120,000 per IMP per year. Dividing this over the number of active hosts leads to a cost per Arpanet host of approximately $50,000 to $60,000. This cost is invisible to Arpanet sites because the costs are picked up on a per-IMP basis directly by the Arpanet sponsoring agencies.

CSNET Nameserver Project. The CSNET Nameserver, developed at the University of Wisconsin, contains information about CSNET users and other data of interest to the CSNET community. In phase one of the Nameserver project, CSNET users maintain their own database entries and do keyword lookups to find netmail addresses and other information about users. The CSNET Nameserver project is now entering the second phase with the introduction of automated lookup. In this phase, a user defines a personal alias for an address to which he wishes to send a message. This alias is not bound directly to a specific address. The user mail software, in attempting to resolve the alias, generates an inquiry to the nameserver database, which responds with the current correct address of the intended recipient. This address is substituted into the message header and the message is passed to the mailer. For a CSNET site with a high-speed network interface, this takes place at the time of message submission; for a PhoneNet site, automatic PhoneNet mail is exchanged to effect the resolution.

New Mail Software. A major rewrite of MMDF, the CSNET standard mail transport system, fully supports the new Internet address standard set forth in RFC 822. In addition, it has been extensively reworked internally for efficiency. The old version of MMDF used to maintain a single queue of messages for all sites, including both Arpanet and PhoneNet. MMDF sorts the outbound queue each time it attempts delivery to a channel (i.e. to a PhoneNet site), which can cause significant delays in the case of an overly large queue. The new version maintains a separate queue of messages per channel, which cuts queue-sorting time significantly.

Member-contributed facilities exist for connecting MMDF to UUCP, including the path-alias mechanism. This allows RFC 822 addressing to be used on all messages, which will then be routed into the appropriate transport domain for processing. Another member-contributed facility supports the handling of large mailing lists.

Software Support. CSNET is committed to tracking Berkeley UNIX. Transporting and testing the software will be completed in early 1984. As part of a separate charter for network support, BBN is now in the process of replacing the Berkeley TCP/IP implementation in 4.2BSD with BBN's own SYS.10-descended TCP/IP implementation. SYS.10 is the TCP/IP version that most 4.1BSD sites are now running. After this has been tested, they will proceed to install the buffer management schemes which give 4.2BSD its high performance on local networks. At that point a network "bakeoff" will be held to determine which implementation will be supported by DARPA in the future. CSNET plans to use the most stable version available, given its charter as a service organization.

Supported releases of X25Net and Nameserver software will be available for 4.2BSD only. PhoneNet software will be available in C for V7, 4.1BSD and 4.2BSD versions of UNIX, and (using member-contributed software) VMS and other systems which support Pascal.

Network Reconfiguration. The physical realization of the PhoneNet is undergoing revision. Currently, there are two CSNET PhoneNet relays: one on the East Coast at BBN, and one on the West Coast at the Rand Corporation. CSNET is in the process of consolidating the operation of the Relays. The West Coast Relay will shortly move to BBN as well, which will place all four CSNET machines (Service Host, CIC Host, and both PhoneNet Relays) at BBN. This step has been taken because it has been observed that it is cheaper to run a single computer center, and communicate via WATS lines, than to run multiple centers.

Mail Gateways. Negotiations are under way to connect CSNET with several European and domestic networks. There are already several affiliate members in Canada in the process of connecting to CSNET. An organization in Sweden is soon to become a gateway to the Swedish University Network using the X25Net facility. Telenet permits direct connection with many of the European X.25-based Public Data Networks. For instance, the UCL-CS gateway machine in England may be directly accessed from any X25Net machine.

In addition, a CSNET/BITNET gateway will be operational in the first quarter of 1984. BITNET is a network of about 80 sites, including primarily IBM and UNIX systems, connected via dedicated phone lines.

Software Administration Over Computer Networks — The Exptools Experience

Joseph L. Steffen
AT&T Bell Laboratories
Warrenville-Naperville Road
Naperville, IL 60566

Software Administration over Computer Networks--The Exptools Experience

Joseph L. Steffen

AT&T Bell Laboratories
Naperville, Illinois 60566

1. INTRODUCTION

This abstract describes a working method for large-scale, distributed administration of software over computer networks. It is being used at AT&T Bell Labs for the experimental tool package (exptools), a collection of the latest programming and text processing tools supported by the authors or other individuals rather than an official organization. It contains more than 60 tools maintained by over 20 people on more than 100 machines of four types connected by three networks of two types. There is an overall coordinator who controls the installation of new tools to insure a uniform tool set across all machines; but the individual tool supporters actually maintain the tools, that is, fix bugs and install new releases.

The novel features of *exptools* administration are:

- The division of software administration between overall coordination and individual tool support.

- The use of existing computer network capabilities (file transfer and remote command execution) to install and update tools.

The latter avoids manually logging into each machine, which has several benefits:

- A tool can be updated on all machines in hours instead of days or weeks.

- Adding more machines does not significantly increase administrative effort.

- Tools can be supported by people or organizations outside the computer center because the coordinator has final control.

- Work done by the coordinator is minimized, which allows one person to administer scores of machines.

- Tool administration can be separated from machine administration, i.e., the local system administrator need not install or update tools.

2. BACKGROUND

Our site has about 130 machines, so we needed a mechanism for installing the tool package easily on a new machine. We decided to have an *exptools* login on each machine that would contain *bin*, *lib*, and *man* directories for the tools' executable files, manual pages, and auxiliary files, respectively. With all the files under one directory structure they could be easily gathered by the *find* command and copied to another machine.

Since the tools are not recompiled for each machine they must be machine and file system independent. This means that they must use *uname* to find the machine's name if needed, and cannot have hard-coded path names for their auxiliary files.

So that tools do not have to search /etc/passwd to find the tools login's home directory, an environment variable (TOOLS) can be set to this directory. This is best done in the system profile (/etc/profile), but can also be done in each user's profile ($HOME/.profile). (A tool is provided to change the user's profile to set this variable and also to add $TOOLS/bin to the user's PATH.)

These restrictions require changes to some tools before they can be installed, but the use of a login to contain the tools' files has some compensating advantages:

- The login can be moved to a file system with more space if needed.
- Since *root* does not own the files the damage possible by unauthorized access is limited.
- The executable files are portable to all machines of the same type.

The latter means that when tools are added or updated only the executable files are shipped to each machine and installed, instead of shipping the a tool's source code to every machine and compiling and installing it. The executable files are generated by shipping the source to one machine of each type, compiling it, and shipping the executable file back. There is no CPU time spent compiling on nearly all machines, and since the source is usually larger than the executable file there is less network traffic.

3. IMPLEMENTATION

The first requirement of our method to administer software over a network is the ability to send commands along with a file over the network, something like the *uux* command does for the *uucp* network.

The next requirement is a *setuid* program the target machines that sets the effective user ID (uid) to the tools login on execution. This program is executed by a command sent over the network from an administrative tool.

The command sent with the file has a password parameter that is checked by the *setuid* program. This is used to provide two levels of remote access to the tools login. Tool supporters are given a password that only allows the file sent to replace an existing file. The coordinator has a password that allows the file sent to be installed as a new file, or to be executed as a shell script. The latter feature allows files to be removed remotely, directories to be created, etc.

Tool supporters need three administrative tools. The first (rmake) sends the tool source files to one machine of each type, compiles the tool, and sends the executable files back. The tool supporter then uses the second tool (repfile) to replace the old executable files with the new on the appropriate machines. The tool's other files are also updated with *repfile*. A third tool (nuex) sends commands, not to the *setuid* program, but to an ordinary shell on the remote machine. This gives the tool supports read-only access to the tools login so they can use the *ls* command to check the status of their files. The results of these tools' remotely executed commands is mailed back.

These administrative tools use a list of machine names and types maintained remotely on each machine by the coordinator. This list is updated (with *repfile*) every time the coordinator installs the tools login on a new machine.

The coordinator needs a tool (sendall) to copy the tools login to a new machine, a tool (addfile) to add files to existing tool logins, and a tool (rex) to execute miscellaneous shell commands to remove files, for example. There are also tools built on these for other coordinator functions:

- Adding a directory (rmkdir).
- Adding all the files in a new directory (adddir).
- Auditing the contents of the tool logins (audit).

Finally, three tools are provided for the users to get

- News of new and enhanced tools (toolnews).
- Manual pages and user guides (toolman).
- The current list of tool supporters and the tool coordinator (providers).

4. EXPERIENCE

Remote installation of tools by the coordinator required considerable ingenuity to get around shell commands and system calls that would not work when executed by the *setuid* program. The *cp*, *mv*, and *ln* commands are coded such that the real *uid* must be the tools login. The *Cat* and *rm* commands were used to simulate *cp* and *mv*. However, a file link could only be created by using the *link* system call from the *setuid* program.

The *mkdir* command uses the *mknod* system call, and the *rmdir* command uses the *unlink* system call. Both require that the effective *uid* be *root*, and that the real *uid* have write permission for the parent directory. Finally, the *mkdir* command makes the real *uid* owner of the new directory.

When commands are sent over a network and executed on a remote machine the real *uid* will be the *uid* of the owner of the network driver, usually a special network login like *uucp*. The effective *uid* can be changed to the tools login by executing the *setuid* program, and to *root* by executing commands that are owned by *root*, like *mkdir*. So to make a directory remotely, send a set of commands to

1. Execute the *setuid* program (effective *uid* becomes the tools login).

2. Use *chmod* to change the mode of the parent directory to writable by others.

3. Use *mkdir* to make the child directory (effective *uid* is *root*).

4. Use *chmod* to change the mode of the parent directory back to the original (effective *uid* is the tools login).

5. Exit the *setuid* program (effective *uid* becomes the network login).

6. Use *chown** to change the ownership of the child directory from the network login to the tools login.

Removing a directory requires a similar script of commands.

Updating an executing file is non-trivial because it cannot be opened for writing if it is a pure procedure (shared text) file. It can be updated by linking the file to a temporary name, unlinking the real name, and installing the new file with the real name and the old file mode.

If a file update fails for any reason (such as file system full) the old file is restored by unlinking any partially created file and linking from the temporary name back to the real name.

Most operational problems encountered were network-related. The network would go down, jobs would disappear, or worst yet, files would be truncated on delivery. To combat the latter, the administrative tools were changed to check the length of the file sent against the length of the file received. Subtle differences in the four versions of networking software at our location also complicated the administrative tools.

The use of passwords for remote file updates has provided enough security for our corporate environment, but may not be enough for a university environment. We are looking into replacing the passwords with digital signatures.

* *Chown* is a command in System III and V, but not in 4.1BSD.

While learning the tool update password does not give you access to more that the files owned by the tools login, there is always the danger that a Trojan horse could be put in a tool, so system administrators are told not to use them when super-user.

5. RESULTS

This method of remote administration of software over computer networks allowed *exptools* to expand from the initial 21 machines to over 100 machines in less than two years, without overburdening the coordinator or tool supporters.

Many tools are heavily used and are considered an essential part of the computer environment, so people want them on all the machines they use, hence the dramatic spread of *exptools*. The lack of computer center involvement has become an asset because the tool package can be installed on a private machine without implying computer center support.

One result of our success was that we had to convert the on-line manual pages from unformated nroff source to formatted output because so many people were requesting them that formatting them was overloading the machines.

6. THE FUTURE

Ideally, the best tools will pass from the experimental stage to official support by the computer center or become an official UNIX product. This has already occurred for a few tools.

We have a trial underway for supporting *exptools* at other AT&T Bell Labs locations over an interlocation network. Initial results have shown the need for changing the administrative tools to allow for a local *exptools* coordinator at each location. The local coordinator controls the local tool set, but updates to common tools are done remotely.

7. CONCLUSIONS

Our experience with *exptools* points the way to efficient administration of hundreds of desk-top computers connected to local-area networks. This removes a stumbling block to their acceptance because many people want desk-top computers, but few are willing to be their own system administrators.

8. ACKNOWLEDGEMENTS

I would like to thank Greg Guthrie for his pioneering work on the precursor to *exptools*, and Mike Veach for his help throughout the development of *exptools*. Thanks to the more than twenty people who give their own time to provide support for the individual tools, and especially to Tom Clark for taking over as coordinator.

A Distributed File System For UNIX

Matthew S. Hecht, John R. Levine, Justin C. Walker*
Interactive Systems Corporation
8689 Grovemont Circle
Gaithersburg, MD 20877
and
*441 Stuart Street
Boston, MA 02116

A DISTRIBUTED FILE SYSTEM FOR UNIX
(Extended Abstract)

Matthew S. Hecht
John R. Levine*
Justin C. Walker

INTERACTIVE Systems Corporation
8689 Grovemont Circle
Gaithersburg, Maryland 20877
(202) 789-1155
allegra!ima!matthew

and

*441 Stuart Street
Boston, Massachusetts 02116

November 1983
revised February 1984

Summary

We describe a design for achieving user-level transparent access to remote files in a local area network of homogeneous UNIX# systems. The design features (1) a *remote mount* system call that allows the mounting of a remote directory onto a local directory; (2) a specialized *remote function call* mechanism that allows one UNIX kernel to call functions in another; and (3) a connectionless scheme for communication between hosts. This note describes work in progress.

1. PROBLEM STATEMENT

The problem we solve here is how to make access to local and remote UNIX files (in a local area network) indistinguishable to users; that is, the syntax and semantics of local and remote file access are identical. For example, the user of a command like

```
cp x y
```

where x and y are arbitrary pathnames, can be oblivious to the location of files x and y (one or both may be local or remote) since the underlying UNIX system calls do the right things in either case.

Transparent access to remote files in a local area network of UNIX systems provides new opportunities for file sharing. Independent UNIX systems can share files, diskless workstations can obtain file service from another UNIX system, and workstations with low capacity disks can share databases. Transparent access also allows files to be relocated without breaking programs.

Our solution features a remote mount system call, and roughly places the interface to remote files at the i-node function level of the kernel. This work contains a novel mix of implementation ideas, yielding a simple, clean, and practical solution. Instead of assigning a remote file-server process to a local user process, we use a pool of kernel file-server processes that feed from a

UNIX is a trademark of Bell Laboratories

common request queue. In addition, we use a connectionless (datagram-based), specialized remote function call mechanism that draws ideas from work by Nelson [6].

Extant work on distributed file systems for UNIX is extensive and growing; we comment on a few related papers here. Chesscn [1] attributes early work on remote mount to Lucas and Walker [5] at National Bureau of Standards. Glasser and Ungar [2] at Bell Labs studied a read-only, connectionless datagram scheme for remote mount. Our work is similar to that of Luderer and others [4] at Bell Labs on S-UNIX/F-UNIX and to a design of Plexus Computers called Network Operating System (NOS) by Picard described by Groff [3]. However, these papers indicate a different process architecture with communication based on virtual circuits. Also, the S-UNIX/F-UNIX work makes the cut at the system-call interface, not at the i-node interface. The COCANET project of Rowe and Birman [8] at UC Berkeley is more ambitious than our work; we do not handle remote processes. The LOCUS project of Popek and others [7, 9] at UCLA is also more ambitious; we do not consider replicated files nor transactions nor remote processes. To our knowledge, the only above-mentioned projects in operation now are NOS and LOCUS.

2. NETWORK NAME SPACE MODELS

Although various models exist for extending the UNIX filesystem name space to a network, we have chosen the remote mount model because it has properties that we desire, such as location transparency and ability to specify a remote root file system. The models include:

- Host- and ~~Path~~Route-Qualified Absolute Pathnames
- Global Root (Super-Root)
- Per-Host Subdirectories
- Symbolic Links
- Remote Mount
- Combinations of the above

We explain these models with familiar directed graph terms. To get started, it is convenient to pretend that a single UNIX file system is a rooted, connected, directed tree (typically pictured as a triangle), with arcs directed from the root. Recall that an arc is a pair of points (tail, head) drawn as an arrow from tail to head (associate "head" with "arrowhead"). A single UNIX file system is not really a tree because of parent ("..") entries and links.

The various network name space models explain how to draw a network filesystem tree, and sometimes suggest implementations. *Uucp* has host- and ~~path~~route-qualified absolute pathnames. Picture separate trees, one for each host system. A file name may consist of a host name followed by the character '!' followed by an absolute pathname, in which case the pathname is sent to that host for resolution. Or, a path of '!'-terminated host names specifying a route may precede an absolute pathname.

The super-root (global root) model makes each existing root file system a subdirectory of a new super-root. A super-root qualified pathname can begin with a special character, say '@', followed by a host name, in turn followed by an absolute pathname.

COCANET [8] supports per-host subdirectories, where the root of each host, in addition to its local subdirectories, has a subdirectory for each host, including itself.

Symbolic links [11] generalize normal, intra-filesystem UNIX links. A special i-node type, called a symbolic link file, contains a pathname. When the name resolution function *namei*() encounters this file while resolving a name, the contents of the symbolic link file are prefixed to the rest of the name (if any). If the symbolic link file contains an absolute pathname, the resulting name is interpreted relative to the root directory. Symbolic links allow inter-filesystem links on the same host. In addition, symbolic links to directories are permitted by a superuser. Consequently, we can have network links if we support a symbolic link file that contains a host- or route ~~path~~-qualified absolute pathname.

The remote mount model generalizes the (normal) mount model. The *mount* command allows us to attach a tree, the root of another file system, onto a directory of our root file system. The *rmount* command allows us to attach a remote tree, a directory in a remote file system, onto a directory of our root file system. We can mount the tree with read-write access or read-only access.

In addition, combinations of the above models are possible. For example, we can use either symbolic links or remote mounts or a combination of the two to provide per-host subdirectories and to customize the resulting name space.

In this note we omit a discussion of the problems, advantages and disadvantages of each model. However, we do point out that interhost linking (by symbolic links or remote mounts) is potentially unsafe because it can introduce unsafe loops in the network tree that can result in undesirable behavior by tree walkers like *find, du,* and *cpdir* (cp -R). Also, commands like *mvdir* may not preserve loop-freedom. There are regimes of constraints for interhost linking that are both safe and flexible in practice.

3. SKETCH OF DESIGN

Our design consists of modifications to the UNIX System V kernel.

3.1 System Calls

We have added several new system calls, including

> rmount(hostname, rdir, ldir, ronly), and
> rumount(dir).

Rmount mounts remote directory *rdir* of system *hostname* on local directory *ldir*, where *ronly* specifies read-only or read-write access, and *rumount* unmounts a remote directory from the given local directory. The *rmount table*, analogous to but separate from the mount table, helps the kernel function *namei*() cross rmount points. A successful rmount installs a TO entry in the local rmount table and a FROM entry in the appropriate remote rmount table.

Figure 1 shows an example of a remote mount where directory /g of host green is mounted onto directory /b of host blue. Now, /b/w/z is a valid pathname on host blue.

3.2 Client (Request) Side

We distinguish i-nodes that represent locally stored files from i-nodes that represent remotely stored files. The latter we call *surrogate i-nodes*. If a pathname crosses an rmount point, the local system sends a *namei*() request with the remainder of the pathname to the remote system. The remote system resolves the pathname, calls *iget*() there, and sends, as the *namei*() response, a *handle* that identifies the remote i-node. The requesting system uses the (host, handle) pair to define a surrogate i-node, which it installs in the local *i-node table*. Only the local *file table* is used during access to a remote file, not the remote file table.

Operations on surrogate i-nodes generally translate into remote function calls. Currently, these operations include *access(), chmod(), chown(), closef(), ioctl(), iput(), itrunc(), iupdat(), lockf(), link(), maknode(), namei(), openi(), owner(), plock(), prele(), readi(), rmount(), rumount(), seek(), stat1(), unlink(), utime(), writei()*, and a few others. In addition, we are experimenting with more comprehensive operations to lessen network traffic. A new stub-manager module, which we call the *agent*, consolidates the remote function call mechanism for the client, and hides request/response message formats from caller modules.

3.3 Server (Response) Side

Incoming remote requests for local file service are enqueued on a common request queue. A pool of kernel processes handle remote requests to local i-nodes. The code for a server looks like:

```
for (;;) {
        wait for a request;
        dispatch function;
        send response;
}
```

3.4 Transport Protocol: Packet Exchange Protocol

As the transport protocol, we currently use the Packet Exchange Protocol (*PEP*) [10], since it provides most of the request/response model assumed by our design. This protocol uses a *socket* abstraction, a network address where processes can send packets and rendezvous, and a *packet* abstraction, a message container.

Figure 2 shows the software layers in the current prototype, and Figure 3 shows innards of an implementation of the PEP layer. In Figure 3, a circle denotes a process, a rectangle denotes a module, an arrow denotes a function call, and a "+" denotes missing details. Because PEP is not well-known, we include a brief description of it here.

PEP has a three-function interface. *WaitRequest()*, called by a server, waits at a known socket for an incoming request packet. *SendRequest()*, called by the agent, transmits a request packet to a known server socket, and blocks until it returns either a response packet or error. If necessary, the packet is retransmitted a specified number of times, waiting a specified interval between retries. *SendResponse()*, also called by a server, transmits a response packet to a socket and does not block. The caller of *SendResponse*(), a server here, is responsible for detecting duplicate requests and retransmitting responses.

Network layer code enqueues arriving PEP packets onto the PEP receive-queue, and wakes up the PEP receiver process. This process dequeues packets, matches them to *SendRequest*() and *WaitRequest*() entries in its match table, and wakes up the appropriate sleeping server or client process.

4. PROBLEMS ADDRESSED

In this section we give an abbreviated description of various design problems that arise and indicate solutions for some of these.

4.1 Rmount Implementation

Problem areas that arise in implementing the rmount model include multiple hops, "..", rmount loops, and security. We comment only on the first two.

In a multiple-hop design, messages are transshipped through intermediate hosts. A surrogate i-node on host H1 may point to a surrogate i-node on host H2, which in turn may point to the real i-node on host H3. However, it is possible to eliminate multiple hops and produce a single-hop design where messages are sent directly to the end host. In a multiple-hop design, *namei*() can cross an rmount point in a server by passing the remaining pathname to the next host. In a single-hop design, when crossing an rmount point a server *namei*() can return the identity of the next host and the pathname progress to the original client host, which sends the remaining pathname to the next host. Thus, we can stack the host path walked by system calls *chdir* and *chroot*, and pop retraced subpaths.

To ascend (with "..") an rmount point, we remember the previous jump-off point of this *namei*() and use a FROM entry in the server rmount table. Thus, we can allow more than one local directory per host to rmount the same remote directory. With a multiple-hop scheme, we can reflect the remaining pathname back to the requestor host at the jump-off point, which is sent in the original request. With a single-hop scheme, we can stack the rmount jump-off points and uniquely retrace "..".

4.2 PEP Model

PEP, while useful here, does not exactly match our application of remote and nonidempotent function calls. PEP is a request/response model for idempotent requests. While *idempotency* in this context normally means that reevaluating a duplicate request produces the same value and state as evaluating the original request, we can relax it to mean that request reevaluation is harmless, as in the case of a time server request. The function *namei*(), for example, is nonidempotent: if it locks an i-node the first time called, then the server process will block the second time called as UNIX does not remember the identity of the locking process. Consequently, the application layer RF solves some protocol problems that perhaps belong to a transport layer, like duplicate request detection.

4.3 Process Architecture

Three problems associated with our process architecture are side-stepping the notion of a session, cleaning up server state upon actual or presumed death of a client host that holds server host resources, and preventing server deadlock. We omit discussion of these here.

4.4 Miscellaneous

To avoid unnecessary packet copying, one can add a little slop space before a buffer to hold protocol headers, or use a scatter/gather data structure that network devices support.

5. CONCLUSION

The current design has limitations that. while not insurmountable, need mention. First, we consider a network of homogeneous hosts so that an executable from a remote host runs on the local host. With heterogeneous hosts, this is not the case. Second, we consider a partitioned, but not replicated, file system. Third, we do not consider remote processes, executing a command on a remote host, functionality we plan later. These intentional limitations let us chew the first bite.

Based on our experience to date with a prototype of this design, our conclusion is that a connectionless rmount approach for a distributed UNIX file system is practicable. More performance experience and code tuning is needed before we can remove the "b" and "e" from that last word.

Acknowledgment

We gratefully acknowledge helpful discussions and comments on the ideas in this paper by Lee Cooprider, Dan Franklin, Gary Gordon, Brian Lucas, David Marx, and Joe Sokol.

References

[1] Chesson, G., Borden, B., and Gurwitz, R., "UNIX Systems on Local Area Networks," tutorial, *1984 UniForum Conf.* (January 1984).

[2] Glasser, A., and Ungar, D., *Fifth Berkeley Workshop on Distributed Data Management and Computer Networks* (February 1981).

[3] Groff, J.R., "Modified UNIX System Tames Network Architecture," *Electronics*, pp. 159-163 (September 22, 1983).

[4] Luderer, G.W.R., *et al.*, "A Distributed UNIX System based on a Virtual Circuit Switch," *Proc. 8th Symposium on Operating Systems Principles*, Pacific Grove, Calif., pp. 160-168

(December 1981).

[5] Lucas, B.W., and Walker, J.C., unpublished work, National Bureau of Standards (1976).

[6] Nelson, B.J., "Remote Procedure Call," report number CSL-81-9, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, Calif. (May 1981).

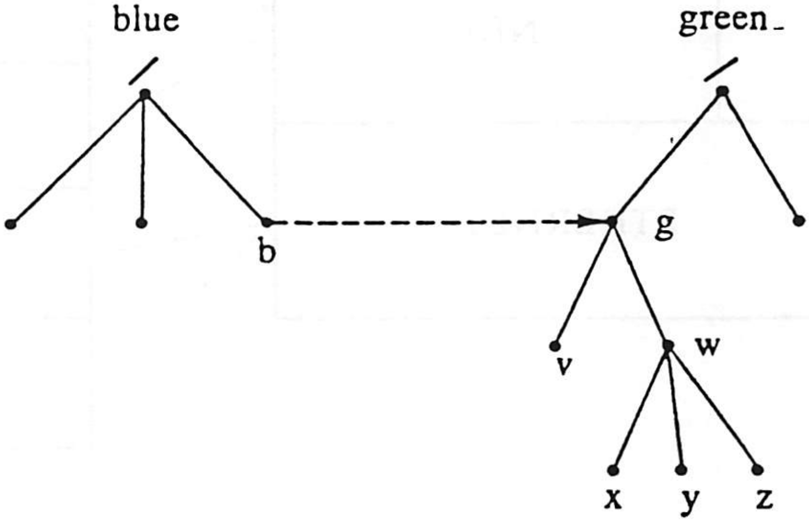
[7] Popek, G., *et al.*, *Proc. 8th Symposium on Operating Systems Principles*, Pacific Grove, Calif., pp. 169-177 (December 1981).

[8] Rowe, L.A., and Birman, K.P., "A Local Network Based on the UNIX Operating System," *IEEE Trans. on Software Engr.*, Vol. SE-8, No. 2, pp. 137-146 (March 1982).

[9] Walker, B., *et al.*, "The LOCUS Distributed Operating System," *Proc. 9th Symposium on Operating Systems Principles*, Bretton Woods, New Hampshire, pp. 49-70 (October 1983).

[10] "Internet Transport Protocols," Xerox System Integration Standard XSIS-028112, Chapter 8, pp. 49-51, Xerox Corporation, Stamford, Connecticut (December 1981).

[11] *UNIX Programmer's Manual*, 4.2 Berkeley Software Distribution, Univ. of Calif., Berkeley, Calif. (August 1983).



```
at blue: rmount [-r] green /g /b
```

Figure 1. Example of a Remote Mount.

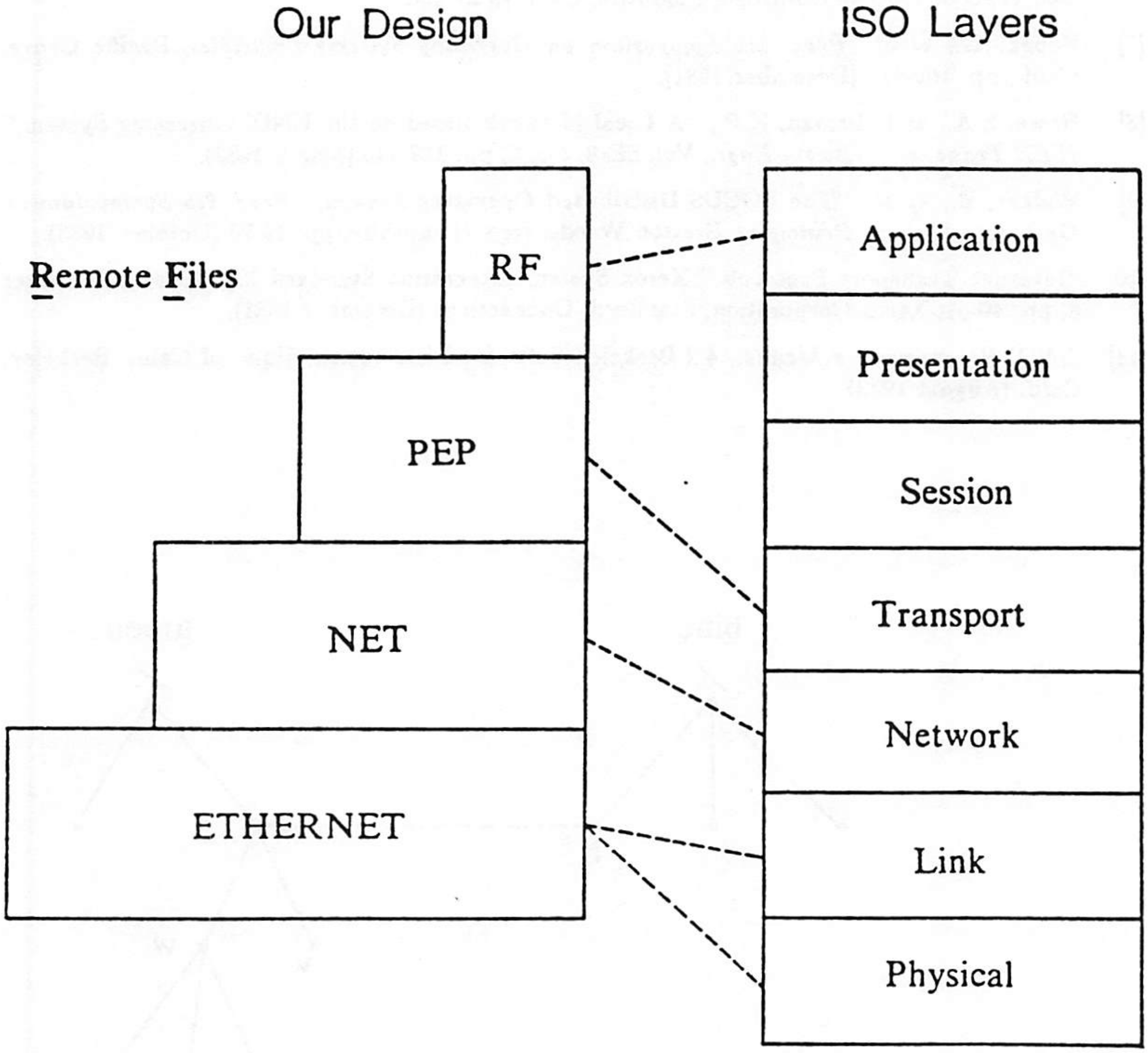


Figure 2. Software Layers in Prototype.

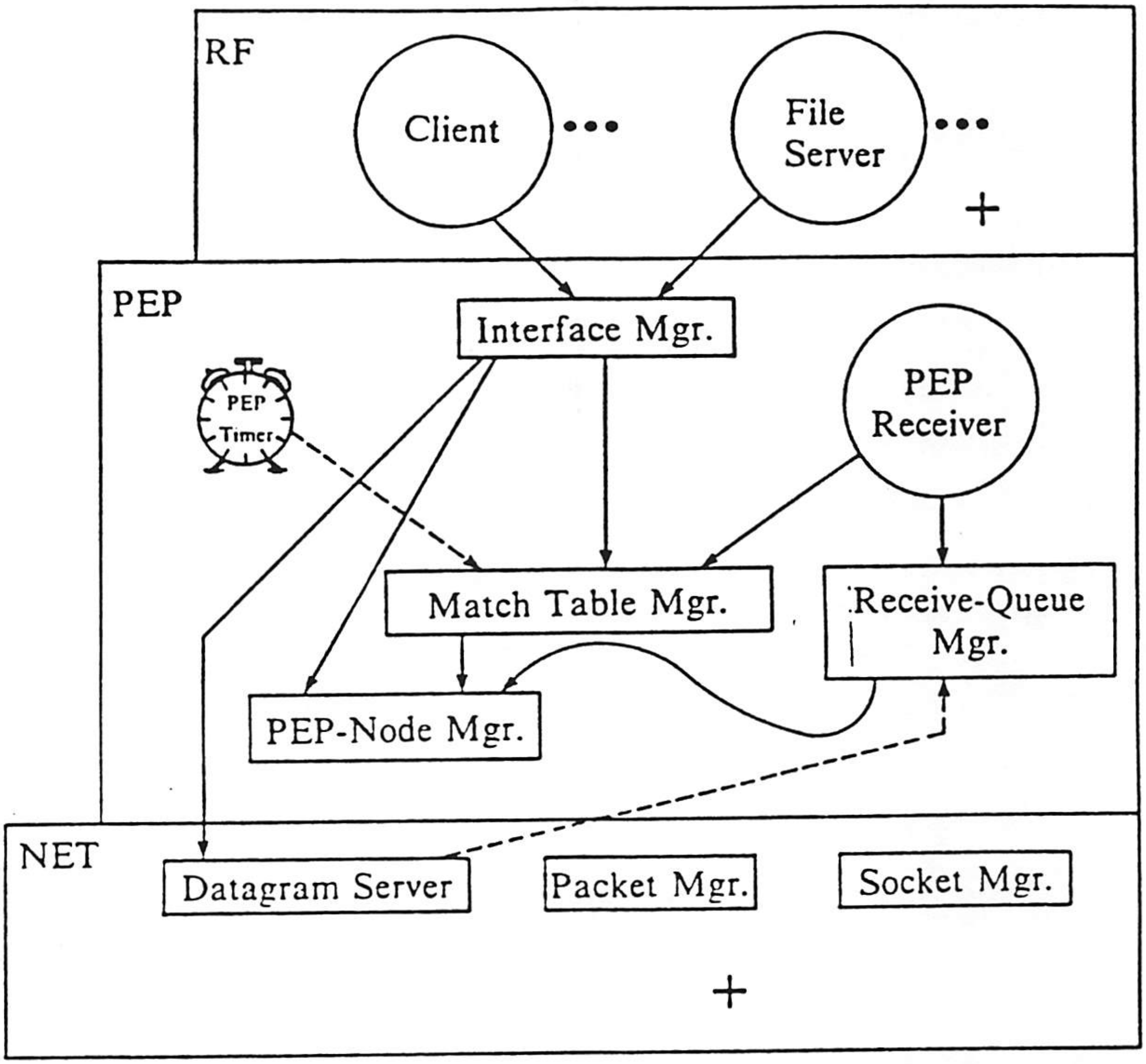


Figure 3. Packet Exchange Protocol (PEP) Innards.

Multiprocessor Debugging on a Shared Memory System

Chet Britten, Paul Chen
Metheus Corporation
5510 N.E. Elam Young Parkway
Hillsboro, OR 97124

MULTIPROCESSOR DEBUGGING ON A SHARED MEMORY SYSTEM

Chet Britten
and
Paul Chen

Metheus Corporation
Post Office Box 1049
Hillsboro, Oregon 97124
decvax!teklabs!ogcvax!metheus!chet

ABSTRACT

This paper discusses a simple debugging technique used in developing a shared memory multiprocessor MC68000[1] UNIX[2] operating system. Techniques applicable to multiprocessor debugging in general, and to debugging on specific hardware, are presented. Some of the problems and trade-offs are also covered. In developing the operating system for the Metheus λ750 VLSI design workstation, we first developed several tools for debugging the kernel. Most of the ideas offered could be used by any implementation. Hardware design considerations to make debugging easier are also discussed.

[1] MC68000 is a trademark of Motorola

[2] UNIX is a trademark of Bell Laboratories

Multiprocessor Debugging

Introduction

The project we worked on at Metheus was a UNIX based workstation for VLSI design, the λ750. We have implemented 4.1bsd with 4.1c Ethernet[3] , and virtual memory. We have high speed color graphics on our system and a multi-window implementation.

There are three Motorola MC68000's ("68K's") and a bitslice processor in the Metheus λ750. There are also six processor based controllers. The u68K executes the UNIX kernel and application programs. The c68K runs its own real-time kernel and takes care of memory management, I/O control, and the debugger. The d68K is responsible for the graphics display list processing. The bitslice does the actual drawing and filling on the monitor. The extensions to our debugger for multiprocessor debugging were added in one day.

Fundamental Debug Functions

Before we added multiprocessor capabilities our debugger supported the following functions:

Breakpoints
: There can be multiple breakpoints set symbolically. All of the following functions can refer to memory as offsets from source file routine or data names.

Trace
: Execute one instruction, then stop and display registers and the disassembled current instruction.

Execution
: Start or continue execution at an address.

References
: Memory or I/O can be examined or changed as bytes, shorts, or longs.

Displays
: The debugger can display data in decimal, hex, octal, or as 68000 instructions.

Call stack
: The C stack frame, or calling sequence, can be displayed. This shows routine names, parameters, and return addresses. Having the return address displayed makes it easy to break on the return from a subroutine call and to look at the value returned.

Simulated disk
: The debugger can simulate a disk so that we were able to develop the operating systems before the hardware was ready. This was done over a an RS-232 line.

Download
: The debugger can download the object code to be tested over an RS-232 line. This was necessary before we were able to load from disk.

Extending Debug Functions for Multiprocessors

Since the processors in our system share memory, many of the original functions did not need to change at all for multiprocessors. Any function that only accessed memory could remain the same. The biggest efforts were changing the breakpoint and tracing code, and adding the ability to start or stop the processors together or individually.

[3] Ethernet is a trademark of Xerox Corporation

Time Constraints

The time constraints we were under in developing our system were tight. We wanted to have a good debugger but didn't want to spend too long adding features to it.

Our goal was to have the system ready for beta test in twelve months. The system software was done in this time by three software engineers. This software included: a multiprocessor debugger; a real-time kernel on the c68K; the port and partitioning of UNIX to the two 68000's; window management; color graphics support; UNIX applications; and drivers for the disk, magnetic tape, Versatec, and six serial ports.

Parts of the Debugger

The debugger consisted of three parts. The host debugger runs on our VAX-780[4]. The host debugger is responsible for the user interface, the symbol table interfacing which allows the symbolic access, and the building of the C calling sequence. It also handles simulated disk requests and downloading. The c68K debug monitor runs on the c68K and communicates with the host debugger over an RS-232 link. Its jobs are to access the 68000 memory and I/O, set and remove breakpoints, dump and restore registers on a breakpoint or trace trap, and provide single step tracing. The c68K debug monitor communicates with the u68K debug routine, the third part of the debugger, through the shared memory. The u68K debug routine (and any other processor debug routine) only has to dump and restore its registers on a breakpoint or trace trap, say it has done so, then wait to be told to restore its registers and continue execution.

Breakpoint Implementation

Single processor breakpoints are implemented on the c68K by using a 68000 trap instruction. When the c68K hits a breakpoint trap instruction it traps to a routine to dump its registers. It then does a context switch and starts executing the c68K debug monitor. If we want to continue past the breakpoint without removing it, the debug monitor must restore and trace over the original instruction, then restore the breakpoint before continuing execution.

To implement multiprocessor breakpoints, the u68K dumps its registers when it hits a breakpoint trap, says that it has hit the trap, then waits to be told to go again. The c68K debug monitor can then examine the register save area of the u68K, changing any registers if desired. The c68K can place or remove breakpoint instructions for the u68K anywhere it needs to. The c68K then tells the u68K to go and the u68K will resume execution. If we want the c68K to begin execution also, it will see this in a flag when it tells the u68K to go, and will resume its execution at the same time the u68K resumes its execution. This allows us to closely synchronize the starting of both processors.

Trace Implementation

Tracing is very similar to breakpoints. The 68000 has a trace bit in its status register. If this bit is set, the 68000 will execute one instruction, then trap to the code pointed to in the trace trap interrupt vector. The c68K saves its registers on a trace trap, just as on a breakpoint trap, but it doesn't need to worry about restoring the original instruction. On a trace trap, the u68K does exactly what it does for a breakpoint.

[4] VAX-780 is a trademark of Digital Equipment Corporation

Interrupting Execution

We need to be able to interrupt the execution of a processor to find out what it is doing when it isn't going to hit a breakpoint or get a trace trap. The c68K is interrupted by sending a character to a serial debug port. This causes the c68K to get an interrupt, execute code similar to that executed on a break or trace, and enter the debug monitor. The u68K is stopped by having the c68K write to a control port. This control port allows us to start or stop the u68K, examine function codes, and look at the addresses the c68K is accessing. We can tell from the function code whether the u68K is in system or user mode, and with the control port we can step memory cycles. If we need to see the processor registers, we can place a breakpoint trap ahead of where the u68K is executing, then let it go and have it hit the breakpoint as usual to dump its registers.

Expandability

By having the second processor save its registers in a register save area we have an implementation that can easily be expanded. Any additional processors just need to use different save areas. The c68K can then index into these different areas to deal with any of the processors.

Hardware

This implementation was possible because the processors shared all memory. Also, the control port for the u68K allowed us to stop, start, and step memory cycles of the u68K. We were able to interrupt the c68K over a serial port. The 68000 gave us the trace and breakpoint traps as standard instructions. Other multiprocessor implementations could accomplish the same thing as breakpoint and trace traps by saving the original code and patching in jumps to the register save code. For tracing it would need to calculate the offset to the next instruction from the opcode.

Problems

One problem we have because of the completely shared memory is that each processor shares the same interrupt vectors. This meant that we needed a way of letting the processor know where in the indexed register save area it should save and restore its registers. Some of the solutions we looked at were: using different traps for different processors, changing a vector when we know which processor will get the trap, identifying the processor with an unused register or unused field in a register, looking at stack ranges, or a using hardware register which returns the processor id.

Using different traps works well for breakpoints, but not for the trace trap since there is only one trace vector. Also, some systems may not have any available traps for debugging. Another problem here is that if a processor gets lost and is executing where it isn't expected to, it might hit another processor's breakpoint.

Changing the vector works all right for tracing, if we only need to trace one processor at a time. This is a likely case, since any timing problems which would occur because of both processors running at the same time would probably be best caught by breakpoints or a logic analyzer.

Using an unused register or an unused portion of a register limits the program being debugged in its register usage, and may not be compatible with future versions of the processor (using unused bits in a status register for example).

Identifying the processor by looking at its stack range could work, but again there is the problem that we might be lost and have our stack pointer out of the range it's supposed to be in.

A hardware register which would return a unique id for each processor, or some separate memory or vectors seems to be the best solution of those given here.

We chose to use separate traps for the breakpoints, and to change the trace trap vector since we only had two processors. It is likely that a system with more processors would have some separate memory.

Improvements

In our host debugger running on the VAX-780, the user specifies which of the processors to deal with, and then any breakpoints set will be set with the breakpoint appropriate for that processor. Also, the tracing commands refer to that processor. For multiple processors an additional syntax might be to give the name of the processor in the breakpoint command. This would be useful for setting many breakpoints on many different processors. Also, a convenient user interface could know which processor should be executing which code, to save the user from the need to specify this. A helpful improvement would be for the debugger to know about register variable names and local variable stack offsets, as sdb does. We have found ourselves counting bytes quite a bit when debugging Ethernet. Adding the knowledge of structure size on offsets would have helped here.

Conclusion

By having a processor just dump registers, notify that it has done so, wait to go, and restore its registers and go, a simple multiprocessor debugger was implemented. It was well worth the time to implement the debugger. Work is ongoing now on performance monitoring tools.

Adapting UNIX for Data Communications

Charles M. Robins
Rabbit Software Corporation
Malvern, PA 19335

INTRODUCTION

UNIX has come into its own as a standard operating system, particularly for 16 and 32-bit desk top microcomputer workstations. Unfortunately, one of the key elements in the acceptance of microcomputer workstations in the marketplace is the capability of data communications to and from large mainframe computers, minicomputers, and other computers. This is unfortunate because UNIX is not a real time operating system. This paper explores the trends in technology that make it essential for UNIX to allow data communications, and indicates potential extensions to UNIX for data communications, especially on 16 and 32 bit micros.

THE NEED FOR DATA COMMUNICATIONS: TRENDS IN TECHNOLOGY

Data processing actually comprises three fundamental major components: data manipulation (processing), data storage, and data movement. The 1970's saw a dramatic improvement in the first two components. Good examples are VLSI technology and Winchester disks, which have allowed well over a hundred-fold improvement in computing power and local data storage. Together they paved the way for the emergence of the microcomputer workstation. The third component, data communications, did not see dramatic improvement in that decade. In the early 70's data communications rates generally ranged from three hundred baud to ninety-six hundred baud; the same is typically true today. However, new technologies have emerged that are rapidly producing the same orders of magnitude improvement in communications as we have seen in data storage and processing.

In particular, the deregulation of AT & T, the dropping of the IBM anti-trust suit, the emergence of T1 and other digital data transmission lines (which operate at up to 1.5 megabytes per second), fiber optics and local area networking, satellite technology and the space shuttle (which allows satellites to be inexpensively deployed), allow for great improvement in data communication speeds: easily a hundred-fold for long distance communications and often many thousand-fold in local area networks.

Additionally, networking protocols, the software which utilizes these new technologies, have been maturing. SNA, X.25, Ethernet or any of the local area network protocols represent this maturation.

The effect of these developments will be pronounced. Improvement in data communications implies changes to the economics of computing, just as VLSI and Winchester disk technologies changed the economics in the late 70's. Minicomputers, which have inherent weaknesses requiring users to deal with their own backup operations, storage, etc., are likely to be supplanted by locally networked microcomputer workstations connected through increasingly responsive data communications networks to large mainframe computers. The large mainframe computers will handle mass data storage and manipulation, and will provide centralized operations, backup, etc. Local microcomputer workstations will allow dedicated processing without the usual trade-offs of number of terminals versus number and complexity of applications versus performance.

The key element in the integration of microcomputers into local networks and into mainframe computers is the ability to implement data communications and networking software on the microcomputer. In particular, this requires implementation of such software under UNIX. Furthermore, given the increasingly higher speeds of such networks, a major part of the communications and networking software must often work at very high real time speeds, necessitating the use of dedicated processors for data communications. The combination of these two issues brings to light a serious problem in today's UNIX environments.

LIMITATIONS OF THE UNIX FOR REAL TIME PROCESSING

UNIX is not a real time operating system. It is primarily a multi-tasking, time-slicing/time-sharing operating system. There is today no standard transportable way to offer real time processes, in particular communications processes, under the UNIX operating system. The only standardization lies in either utilizing RS232C asynchronous communications or writing device drivers to run under the UNIX kernel. But neither of these approaches allows a standard UNIX operating system to support sychronous communications protocols across the multitude of microcomputers available today without some modification to the UNIX kernel.

UNIX does allow prioritization of tasks, and allows interrupt oriented signals and alarms, but there is no guarantee in the UNIX environment that a higher priority task will always be executed either quickly (in a real-time sense) or definitely before some other task. Furthermore, in order to maintain total transportability across different UNIX dialects, the use of pipe transmissions across processes as a form of interprocess communications (between communications processes and other processes) is both necessary and (relatively speaking) very time consuming.

The UNIX kernel can support a degree of real time processing via "wait on event" calls, which could be used to support synchronous I/O device drivers. But even at relatively slow communications speeds, this places a significant load on the processor; to support several synchronous channels, or even a single channel at high speeds, is infeasible as a long-term approach.

WHAT IS DONE TODAY

Generally most UNIX microcomputers today are equipped with a front-end processor in addition to the main processor that runs UNIX. A typical main processor would be a Motorola 68000 or 68020 or Intel 8086, 80186, or 80286 (Xenix). The front-end processor is generally an 8-bit processor, most typically a Zilog Z80. Usually communications between the front-end processor and main processor are associated with a UNIX special file or device driver that allows access to a port or memory window within the associated processor. Code to be executed on the front-end processor is usually loaded in an executable memory format for the front-end processor (in the Z80 this would mean Z80 assembler code) and then executed. Communications between the front-end processor and the UNIX processor are initiated by signaling back and forth.

The approach creates several problems:

1) There is no transportable mechanism for writing communications software and executing it on most UNIX systems. Each particular installation of UNIX on each microcomputer workstation has slight differences in its implementation of front-end communications, based on the communications architecture and the various processors.

2) The communications software to be executed on the front-end processor must be loaded in executable form. This requires a UNIX communications programmer to cross-compile C code into native assembler for the target processor. Since the native assembler varies with the front-end processor, developers must use many separate cross compilers or write in the front-end processor's assembler.

3) Memory on a typical 8-bit front-end processor is limited to 64K. Also, the speed of the processor is usually limited. Neither of these will suffice for increasingly rich networking architectures (e.g., SNA) nor will speeds be sufficient as standard line speeds increase to well above 19.2 Kb.

4) These front-end processors are inherently not multi-tasking, since they are "naked boards" with no operating systems; this implies that if one is trying to put more than one major process on the front-end processors, then one has to write one's own multi-tasking or time slicing algorithms. This should simply not be the responsibility of a communications software implementor.

To summarize today's situation: to implement communications for the UNIX environment, one must utilize front-end communications processors that are not standard and are not interfaced to transportably. The implementor must therefore do a significant amount of unique work for each machine port. Using a standard vendor supplied processsor is not an answer, because each microcomputer has a different bus structure, etc. And lastly, multi-tasking front-end processors are simply not available. In short, implementation of communication software requires the implementor to step down from the UNIX environment back to the low level 8-bit, single process, assembler code environment of the earliest days of computers. This revives all the problems of enhancement, maintenance, and lack of flexibility that were inherent in the enviroments of the 60's. Clearly, this is not suitable for the future of the microcomputer workstation industry.

WHAT IS NEEDED?

What is needed is an environment where real time and communications processes can be written in C in a separate process without concern for the execution environment (i.e., front-end processor). Execution or compile time options could indicate the process' priority and that it is a "real-time" process.

UNIX would ideally support multi-tasking real time processes with priority interrupt structures such as appear in any typical real time operating system. Also needed is the ability to off-load real time processing to front-end processes (when available) transparently to the communications software developer. If a front-end real time processor were not available then the software should be executable on the main UNIX processor. These options would no doubt vary across UNIX implementations. However, the specifics of which real time processor was being used, the availability of real time multi-tasking capabilities, and the nature of the interface to such processors, would be totally transparent to the software developer. The execution environment might be specified as a separate step, or UNIX itself might automatically establish a run time environment based on an indication that a given process was to be real time.

The keys are transparency and transportability, despite the nature or availability of front-end processors. This would allow software developers to work in C, rather than become assembler programmers working under different interface architectures on each microcomputer. It would allow the development of universally available communications software under UNIX.

PROPOSED EXTENSIONS TO THE UNIX ARCHITECTURE

The following extensions to UNIX are proposed, without specific indication of how these would be implemented.

1) Allow communications processes written under UNIX to be written entirely in C (as if they were to be executed on the main UNIX processor); at compile time the developer could indicate which modules or processes are to be compiled for the front-end processor as real time processes. The UNIX environment itself would choose the appropriate compilers, check for front-end setting of appropriate flags for execution, etc. Then, when such a process is fork executed or otherwise executed, it would be automatically loaded and run in the front-end processor.

2) There should be a standard for a front-end processor multi-task real time executive environment, including its interface to a standard UNIX in a transparent fashion. This executive would include real time process priorities sufficient to allow data communications.

3) As an interim solution to facilitate communications software development, a standard syrchonous port interface should be included in the UNIX standards, allowing a start up with parameters sufficient to allow the basic communications protocols used today (e.g., bisynch, SDLC, HDLC, etc.). Given the changing nature of the communications industry, this would be only an interim step.

In closing, this paper should be viewed as a call to action. UNIX as it exists today has many weaknesses; this is no secret to anybody in the UNIX community. A great deal of work has already gone into correcting the most obvious weaknesses, such as those in system security, user friendliness, etc. It is this author's feeling that the question of real time processing and communications processing is at least as critical as any other issue, and must be adequately resolved in the very near future if UNIX is to maintain a leading role as a "standard" operating system.

About the author: Mr. Robins received his B.S. in Mathematics and Computer Science from Brooklyn College, and his M.S. in Computer Science from the Courant Institute of Mathematical Sciences (New York University), where he was also a full Ph.D. fellow. He has lectured extensively, taught computer science, and consulted with both software and manufacturing firms.

Mr. Robins has extensive experience in software design and systems implementation. He has held senior development, managerial, and technical advisory positions at SEI and SMS, two of the nation's leading software service bureaus. In 1982, he co-founded Rabbit Software Corporation.

Rabbit Software's products, based on the company's Middleware™ systems architecture of portable component software, encompass distributed processing, data collection, handling, and presentation, and end user tools. The company's software is distributed through computer manufacturers, and has been implemented under several UNIX and UNIX-like operating systems.

Connecting a UNIX System to an X.25 Network

Joaquin Miller
Pacific Software Manufacturing Company
2608 Eighth Street
Berkeley, CA 94710

Connecting a Unix System to an X.25 Network

Joaquin Miller
Pacific Software Manufacturing Company
2608 Eighth Street
Berkeley, California 94710

(415) 540-5000

In order to meet the growing demand for communications software for Unix systems, Pacific Software began last year a project to install our X.Dot package in the Unix operating system. X.Dot is a C language source code product which supports the CCITT Recommendation X.25 for connection to a packet switched data network. It supports communication between computers and between a remote terminal and a computer using public data carriers such as Tymnet, Telenet and Uninet in the United States and with the national packet switching networks in Canada, Mexico, Europe and elsewhere.

Computer Networking Options

At present there are three common ways to communicate between computers or between remote terminals and a computer: the circuit switched telephone network, leased lines, and public data carriers. Each method has its advantages and drawbacks.

Circuit Switched Telephone Network

The switched telephone network is ubiquitous and instantly available. With a modem at each end, the user need only place a phone call to establish a circuit and begin communication; when finished, she simply hangs up. The network is completely transparent, so any protocol can be used for communication.

On the other hand, a phone call is not inexpensive, especially if the distance is great, and the user pays for the call as long as the circuit is held, even when not transmitting data. The voice grade lines are also subject to noise, which means errors. Bandwidth is low, which limits the maximum data rate. Some communications protocol is necessary in order to detect errors and a more complex protocol is required to easily recover from errors. Obviously, both ends of the conversation must use the same protocol. A number of incompatible protocols are in use.

Leased Data Line

A leased data line will support higher data rates and may have a lower error rate. It will also cost less per bit transmitted, if it is used heavily. But it costs time and money to install, and must be paid for twenty four hours a day, even when not in use. A communications protocol must still be agreed upon for detecting errors and correcting them.

Public Data Carrier

A public data carrier provides a convenient and economical alternative. Any terminal or computer on a public network may request connection with any other, and often with one on another public network. Public data carriers commonly use a packet switched network.

Rather than dedicating a circuit to each connection, a packet switched network establishes a "virtual circuit" through a network of data circuits which are shared by many users. Because the circuits are shared, a packet switched network can be very economical. A small charge is made for the connect time; then the user is charged only for the amount of data transmitted.

A packet switched public data network thus combines many of the advantages of both the circuit switched telephone network and leased lines. In addition the networks provide a standard protocol which protects against erroneous or lost data and results in a very low rate of uncorrected errors.

However, the protocol required for connection to a packet switched network is complex. The user may implement the required complex software in her computer. The public network will then require certification of the computer and software before permitting connection. Another alternative is to buy special protocol hardware to make the connection, but this is expensive.

X.25 and Unix

There is a standard protocol for connection to packet switched data networks, known as CCITT Recommendation X.25. This protocol us used by public data carriers, such as Uninet, Telenet, and Tymnet in the United States, and the national networks in Canada, Mexico, Europe, and elsewhere.

In 1981 Pacific Software introduced X.Dot, a portable, C language source code product which supports the X.25 protocol for connection to a public packet switched data network. Our development environment is Unix, so the recent proliferation of Unix systems provided us ample encouragement to create a version of X.Dot which would run in the Unix environment. Recently we announced X.Dot for Unix, which is a source code product for connecting Unix systems to packet switched networks.

To install X.Dot in Unix, Pacific was faced with a number of design decisions. We discuss some of them here.

Portability

Since our customers use many different processors and machine architectures, it is important that great care is taken to insure portability of the code. The problem of portability is complicated by the existence of many different versions of Unix. We wanted our product to be suitable for Version 7, System III, System V, for Berkeley 4.1 and 4.2 Unix, and for all the individual variations of these versions.

For these reasons it is important that the product require no kernel changes. Since the Unix kernel is a hostile environment for any program and particularly for a real time program, this was an easy decision to make.

The code must not only be independent of Unix version and of machine architecture, it must work with a variety of C compilers. The code must also be suitable for execution either in the main processor or in a separate communications processor board.

Making X.25 Transparent

The major goal when connecting a Unix system to an X.25 network is making the X.25 connection invisible to existing applications software, while providing access to the full facilites of X.25 for programs capable of utilizing them. (A challenging addition is providing means for existing applications to use facilities like collect calling.)

This goal is frequently called "transparency", but might better be called "disguising".

Possibilities

Three levels of disguise are of interest in the Unix environment. The X.25 virtual circuit can be made to look like:

(1) a sequential file,
(2) a simple serial port able to support uucp, or
(3) a full terminal port.

1) Sequential File.

Making a virtual circuit look like a sequential file is much the simplest disguise to undertake. Unfortunately as a disguise it is not very useful. Little existing code will work in so simple an environment; most existing code will expect to use a terminal device.

By careful choice of properties and faking responses to ioctl calls, some existing applications which do not require the ability to login over the network will run. This option is mostly good for writing new software since it presents the most direct and uncluttered access to the network.

2) A Simple Serial Port.

Making a virtual circuit appear as a serial port only to the extent necessary to run uucico (the active agent of uucp) is a useful disguise, since uucp is the natural protocol for sending work and data between Unix systems on an X.25 network. For this level of disguise it is necessary to use the character buffering facilities of Unix and write a simple tty handler which can ignore all the elaborate features of the usual terminal handlers.

An alternative to this approach is implementing level (1) (as described above) and modifing uucp to use the X.25 facilities directly. This has the significant advantage of efficency. Both X.25 and uucp have their own error correcting protocols and when using an X.25 circuit uucp's error protocols are redundant and wasteful.

The disadvantages are that Unix systems not directly on the X.25 network would not be able "dial in" to the network as though they were terminals and use uucp to communicate with systems that are on the X.25 network; new versions of uucp would not be supported; and other protocols would not be supported.

3) A Full Terminal Port.

Making a virtual circuit look like an ordinary terminal port is by far the most useful and difficult way to proceed. The major work is making the "virtual port" behave correctly with all the different modes that can be set via stty and ioctl. This means design work as well as implementation, because of the complexity of the tty modes supported by System III.

This approach amounts to installing PAD (Packet Assembler and Disassembler) software to work with X.25 using a Unix tty handler as model. This kind of PAD behaves like a Unix terminal driver and is not compatible with a CCITT X.3 PAD.

With this approach all the existing Unix applications are supported. More importantly, it is possible to login to the Unix system from an ordinary terminal via an X.25 network such as Telenet, Tymnet or Uninet. Since making a system available over a network is one of the most common uses of X.25, this is the most useful approach.

X.Dot for Unix

Pacific Software has taken an approach which combines 2) and 3). Our goal is full integration of X.25, so as to give the appearance of a switched network. We have developed a product which we call X.Dot for Unix.

The software portability techniques developed for X.Dot have allowed it to be installed on PDP-11, VAX-11, 68000, 8086 and Z8000 systems. X.Dot has been compiled with the Whitesmiths compiler, the standard Unix compiler (Ritchie), the portable C compiler (Johnson), versions and descendants of the MIT compiler, and the Mark Williams compiler. We developed the X.Dot for Unix product on a Dual 68000 Unix system, and recently installed it on a Perkin-Elmer Unix system.

X.Dot for Unix makes available portable C language source code for installing in a Unix system an interface to a packet switched data communications network, and for using this interface from Unix applications.

That X.Dot for Unix is a highly portable product is demonstrated by our experience with Perkin-Elmer. This port involved a new machine, a new version of Unix, a new C compiler, new and unusual serial communications hardware and, since the installation was in Canada, connection to a new network, Datapac. We were able to install and sucessfully test X.Dot for Unix in twenty man days.

Design Considerations

In building X.Dot for Unix, a number of design decisions were faced and resolved. We believe we have arrived at an implementation which is versatile and at the same time provides a simple interface to the user. X.Dot supports the full CCITT 1980 Recommendation, so that additional Unix capabilities can be added with a minimum of effort.

For the disguise to succeed it is necessary to decide how to implement all the necessary X.25 functions such as specifying the address of remote systems and what to do with inbound calls. Further, it is necessary to decide which of the less critical X.25 facilities to make available and how they are to appear to the user. There are a number of significant areas of concern.

1) Network Addressing.

How does the caller specify the network address of the system it is calling? The Unix open command accepts a "file name", but all it passes to the actual device driver is a major device code and a minor code.

We have chosen to use ioctl to send the call connection packet to the interface.

2) Facilities Information.

X.25 lets a caller specify a considerable amount of information when attempting a connection, including things like window size, throughput, collect charging etc. How is a Unix application supposed give this information?

This information is sent in the connection packet via ioctl.

3) Out-of-Band Signalling.

X.25 provides an interrupt level path for passing individual bytes of information. How will these signals be passed?

X.Dot for Unix currently discards this byte and passes the user a break signal. This conforms to common practice in the industry.

4) Resetting the Line.

If a virtual circuit is reset, this creates the problems of signalling an error and deciding how to resynchronize.

Currently we send a break signal to the user. We are working on improved handling.

5) Receiving Incoming Calls.

When a Unix system gets a call over the X.25 network, what is it to do with the incoming information? Unfortunately the X.25 specification does not provide a standard means for addressing from process to process, so this can be a messy design problem.

X.Dot for Unix assumes that such a call is an attempt to login. A user login shell is forked and the call is connected to it. X.Dot for Unix could also examine an optional facilities field in the call connection packet and fork a file or mail server program.

6) Errors and Network Status.

How can the Unix program be informed of problems and other status information about the connection?

Network errors are returned through the standard Unix error facilities.

7) Datagrams.

The fast select facility X.25 provides for the exchange of single packets of data between network hosts without establishing a connection. There is no obvious analogue in Unix.

Unix for X.Dot does not support fast select at this time. It would be possible to pass these packets to a specified, always active, process. This could be a mail server or possibly a remote inquiry application.

8) Partitioning the Code.

A major question is how to install the code into Unix. It is obviously desirable to not modify the Unix kernel. Since Unix does not provide support for real time processes nor for interprocess communication, except by pipes between processes of the same user, we have placed the X.25 code in a deamon process, and the users communicate with it through a device driver.

This solution puts the load of processing the X.25 protocol on the main processor. When higher data rates are desired, it is better to partition the code between an auxilliary processor and the Unix kernel. The main determinate in deciding how to make the partition in this case is the number of active virtual circuits to be supported. Each active circuit costs buffer space. The number of buffers that must be kept for each circuit is equal to the Packet Level window size plus one. Buffers can range in size from 160 to 1060 bytes depending on the network and application.

This may present problems if an auxillary processor is used. The packet level buffers may use more buffer space than is available on a particular auxilliary processor board. This is even more likely to be the case if it is desired to support extended window sizes for communication via satellites. All Packet level buffering and PAD functions could then be handled in the main processor.

Placing all of X.Dot in the auxiliary processor will minimize the processing load on the main processor. If it is desirable to support the fastest possible data rate, only the Frame level should be on the auxiliary processor and the Packet level should be in the main processor (or even on a third processor).

The layered architecture of X.Dot makes any of these solutions easy to implement.

9) Permanent Virtual Circuits.

If permanent virtual circuit are to be supported, the problem is to what process they are to be connected.

The answer to this question is application dependent. Any solution requires permanently active processes. We avoid this problem in the current implementation by not supporting permanent virtual circuit connection to Unix. Of course the X.Dot code itself fully supports permanent virtual circuits.

Conclusion

Pacific Software has dealt with the problem of connecting a Unix system to an X.25 packet switching network by providing a disguise which gives a virtual circuit the the appearance of an ordinary serial port.

X.Dot for Unix provides a combination of the existing Unix communications facilities with low cost, error-free data transmission to multiple systems. At the same time, X.Dot for Unix can serve as a communications front-end for a Unix machine to be used as a timesharing host with terminal access over an X.25 network. With X.Dot for Unix, Unix machines may be connected to public packet switched data networks, or interconnected in a private network for local or long distance communication.

X.Dot for Unix supports many important facilities for the distributed communication application environment. Among these are call-out to or call-in from a remote system, call-in from a terminal on the network to a Unix system, remote file transfer, and remote mail service. The package complements the powerful features of Unix with equally capable and versatile networking.

Additional information on X.Dot for Unix is available from the author:

Joaquin Miller
Pacific Software Manufacturing Company
2608 Eighth Street
Berkeley, California 94710

(415) 540-5000

You CAN Feel Good Knowing It Is Written in C

Alan R. Feuer
Catalytix Corporation
75 Centre Street
Brookline, MA 02146

You CAN Feel Good Knowing It is Written in C

Alan R. Feuer

Catalytix Corporation
Cambridge, Massachusetts

1. INTRODUCTION

Like a growing number of professional programmers, I find the C language to be my language of choice for writing a wide variety of programs. The language is relatively small with few restrictions, the compilers produce compact and efficient code, and the programs, despite the low level of C, tend to be highly portable.

Occasionally people criticize C for its cryptic syntax, its unusual rules for naming data types, or for some of the redundant concepts wrought by the evolution of the language. These are not deep criticisms and they don't hinder the productivity of experienced programmers.

But there is a criticism of C that strikes at its core: *C is a dangerous language*. Let me give you some examples.

2. C IS DANGEROUS

2.1 Arrays are in the eye of the beholder

A programmer can define an array in C, but once storage is allocated for the array, C forgets how long the array is. One consequence is that a program can march off the end of an array with nary a complaint. For instance, declare an array **A** to consist of 5 elements:

```
     int A[5];
```

Print each of the elements of **A**:

```
     for( i=1; i<=5; i++ ) print(A[i]);
```

The *for* loop steps the variable *i* from one to five. But in C the first element of a array is always at index zero. Thus **A[5]** is beyond the end of the array, though C will never tell.

2.2 Functions are very forgiving

Subroutines in C are called functions. As in most languages, in the definition of a function a programmer specifies the formal parameters to the function. Here is the beginning of a function f that takes two integer parameters **i** and **j**:

```
     f(i,j) int i,j; { ...
```

The forgiving nature of functions becomes evident when **f** is called with the wrong type or number of arguments, as in

```
     f(3.4,f())
```

where f is called twice, one time with no arguments and one time with a floating point argument. In neither call to **f** will C complain; in both instances some result will be generated.

2.3 You can write your own I/O library

Input/Output in C is accomplished via function calls, the most common of which is probably the formatted print routine, **printf**. Historically, the flexibility of performing I/O with function calls proved valuable in the evolution of a versatile and efficient library of standard routines. These days, few people need this flexibility, but everyone continues to pay the price. Here is one way that **printf** bites:

```
     int i; float f;
     printf("an int %d and a float %f\n",f,i);
```

prints two values, the first as a decimal integer and the second as a floating point number. **Printf** knows what to print by looking at its first argument, the control string. Following each percent sign is a conversion specification; it tells **printf** both how to print its arguments and what types the arguments are.

But since **printf** is an ordinary function, if it is passed the wrong number or kinds of arguments, no one will ever complain; just strange things will be printed.

2.4 Booleans are mapped onto the integers

One basic feature of C is that every integer can be interpreted as a Boolean value; zero evaluates to false, everything else evaluates to true. The mapping of the Booleans onto the integers contributes to the brevity of the language. For example,

```
     while( s[i++] ) ...
```

is one paradigm for iterating over the elements in the character string **s**. (Character strings are character arrays that end with a zero character. Thus, when **s[i]** is zero, the *while* loop terminates. The ++ following **i** increments **i** by one.)

Unfortunately, with Booleans being integers

```
     int c;
     while( c=getchar()!=EOF ) ...
```

is also a paradigm in C, one in which the variable **c** always has the value 1 in the *while* loop. (The precedence of **!=** is higher than that of =, so **c** gets the value of **getchar()!=EOF**.) The problem here is that it is wholly reasonable to assign the Boolean result of a test to an integer.

Booleans as integers also leads to some interesting (and perfectly legal) anomalies:

```
     3 > 2 > 1 is false, but
     1 < 2 < 3 is true.  On the other hand,
     2 > 1 > 0 is true!  (But then, so is -3 > -2 > -1.)
```

2.5 Pointers can point anywhere

The integration of pointers and arrays in C has both been praised and criticized. The praise derives from the elegance with which arrays and array slices can be manipulated. The criticism is due to the abandon with which pointers can be used. On most machines, any positive integer is a legitimate value for a pointer regardless of whether the value is the address of an object.

One classical result of this libertarian view is the problem of the dangling pointer. Consider a function **tempcopy** that makes a temporary copy of a string:

```
    char * tempcopy(s) char *s; /* return a temporary copy of s */
    {
            char copy[MAXSTRING];
            int i;

            for( i=0; (copy[i]=s[i]) != ' '; i++ ) ;
            return(copy);
    }
```

The string returned by **tempcopy** can presumably be altered without effecting the original, as in

```
            p = tempcopy(original);
            for( i=0; p[i]!=' '; i++ )
                    if( p[i]<'0' || '9'<p[i] ) p[i] = ' ';
```

which replaces each nondigit character by a space. The difficulty is that **tempcopy** puts the copy into a local array that goes away when **tempcopy** returns. Thus instead of pointing to a private copy of **original,** p really points to memory (usually on the call/return stack) that has been freed and that may be used by some other function. The result can be strange behavior far removed from the call to **tempcopy,** such as a later function call returning to the wrong place.

3. WHY C IS LIKE IT IS

I don't want to give the wrong impression. C isn't the product of malevolent designers and its popularity isn't due to masochistic programmers. There are historical and technical reasons for why C is like it is.

Perhaps most importantly, C was designed to be a convenient way of controlling real hardware. From its inception, C has been used for programming operating systems, device drivers, compilers, and other applications where access to the underlying hardware and program efficiency are important. The designers of C intentionally omitted operations from the language that could not be expressed succinctly in the machine language of the time. The machines didn't support strings, so neither does C. The machines treated Booleans as integers and so does C.

C is actually a great step forward in robustness from its predecessor BCPL. In BCPL, a typeless language, all data types needed by a program are pieced together by the programmer from machine words. Early use of C reflected this typeless heritage. Data types were seen primarily as a way of carving up storage and selecting the right operator for an operand. Using types to detect programming errors was only of secondary interest.

The community of early C programmers exerted an enormous impact on the nature of the language. C was not seen as a competitor for languages like Pascal or Ada; it was heralded as a replacement for assembly language. Early users of C were expert programmers that demanded fine control over their programs. The viewpoint taken by the language designers was that the programmer knows more than the compiler; if the programmer wrote it, the programmer meant it.

4. BUT THE WORLD IS CHANGING

With the increasing spread of C, all kinds of programmers now use C for all kinds of programming. The assumption that every C programmer is a wizard is far from true. For most programmers, the flexibility of C can actually hinder productivity by hiding programming errors.

At the same time that C is being used by programmers with less training, it is being used on faster machines with more main memory. The earliest use of C was to write a timesharing system for a half-dozen users on a processor with only a fraction of the power that people now routinely put on their desks. Hence, program efficiency is becoming less of an issue while program correctness and maintainability are becoming more important.

5. WHAT CAN BE DONE?

Given the attractiveness of C as a tool for getting a job done, we are led to ask, "What can be done to blunt C's dangerous points?"

5.1 Lint can help

An important observation regarding C is that a programmer's success with the language is directly related to the programmer's discipline in using it. The first attempt at codifying appropriate discipline for C resulted in the lint program checker. Lint accepts a subset of C that is just as powerful as full C, but that is considerably more robust. It warns against many kinds of questionable type coercions, flags variables that may be used before they have been initialized, complains if a function is used differently from how it is defined, and catches some inherently nonportable constructions. Lint illustrates the variety of checks that can be done at compile time to improve the safety of a language.

But lint is not without its faults:

- Although it can look across source files to check whether the actual parameters in a function call agree with the formals in the function definition, for large programs the time taken to look at every source file makes lint inconvenient to use frequently.

- Although lint can look for questionable combinations of integers and Booleans, the more it reports the more it is ignored.

- And, although it can find some variables that are referenced before they are initialized, since C programs are filled with aliases, lint can't possibly find them all.

5.2 Lint is not enough

Even with the judicious use of **lint**, C is a dangerous language since there are many program errors which **lint** cannot protect against. Here are a few:

- Since the values of most array indexes aren't known at compile time, there is little a compile-time checker can do to detect indexing errors.

- The neat integration of pointers and arrays in C, means that pointers can point anywhere. Like indexes, their values usually aren't known until run time so stray references cannot be detected at compile time.

- Among the objects to which pointers can point are functions. Since lint cannot be sure of the value of a pointer, it cannot check the actual arguments in an indirectly called function against the formal arguments expected by the function.

- Since the control strings to the formatted I/O functions like **printf** and **scanf** can be generated at run time, even if **lint** did know how to interpret the conversion symbols, it couldn't always check to see if the arguments agreed with the control string.

- The values of the arguments to the entry point of C programs, the function **main**, are not known until the program is invoked and hence no checking is possible at compile time.

Fortunately, each of these insecurities can be checked at run time if the compiler is farsighted enough to provide the appropriate information to run-time routines.* With the compile-time checks of **lint** and run-time checks like those above, C programmers can enjoy much of the safety of a language like Pascal, yet still retain the power and versatility of C.

6. CONCLUSION

C has many insecurities. Its arrays have fuzzy boundaries, its functions are forgiving about misuse, its pointers can stray with no control, and its I/O routines can be easily misled. Nevertheless, programmers are attracted to C because it helps to get jobs done.

My intent in writing this paper was not to scare people away from C, but instead to suggest that although C is dangerous, it can be tamed. Traditionally C has been used baldly; C programmers have not had the assistance from their compilers that programmers using other high-level languages have long enjoyed.

The good news is, with discipline in the use of C and with the right tools, C can be safe as well as flexible.

----

* An example of a compiler that generates run-time checks is the Safe C compiler from Catalytix Corp.

An Implementation of the B Programming Language

Lambert Meertens, Steven Pemberton
CWI (formerly Mathematical Centre)
Postbus 4079
1009 AB Amsterdam
The Netherlands

An Implementation of the B Programming Language

Lambert Meertens
Steven Pemberton

CWI (formerly Mathematical Centre)
Postbus 4079
1009 AB Amsterdam

ABSTRACT

B is a new programming language designed for personal computing. We describe some of the decisions taken in implementing the language, and the problems involved.

Note: B is only what the name of the language is called. It is a working title until the language is finally frozen. Then it will acquire its definitive name. The language is entirely unrelated to the predecessor of C.

The programming language B

B is a programming language being designed and implemented at the CWI. It was originally started in 1975 in an attempt to design a language for beginners as a suitable replacement for BASIC. While the emphasis of the project has in the intervening years shifted from "beginners" to "personal computing", the main design objectives have remained the same:

- simplicity;
- suitability for conversational use;
- availability of tools for structured programming.

The design of the language has proceeded iteratively, and the language as it now stands is the third iteration of this process. The first two iterations were the work of Lambert Meertens and Leo Geurts of the Mathematical Centre in 1975-6 and 1977-9, and could be described as both easy to learn and easy to implement. However, there are two sides to ease and simplicity. If something is easy to learn and define, it does not necessarily imply that it is also easy to use. BASIC is testimony to this: it is fine for tiny programs, but for serious work, it is like trying to cut your lawn with a pair of scissors.

The third iteration of B, designed in 1979-81 with the addition of Robert Dewar of New York University, adopts a new characteristic: it is still easy to learn, by having few constructs, but is now also easy to use, by having powerful constructs, without the sorts of restrictions that professional programmers are trained to put up with, but that a newcomer finds irritating, unreasonable or silly. Thus compared to most existing languages that supply you with a set of *primitive* tools, with which you can then build your more powerful tools, B does it the other way round by supplying you with *high-level* tools, which you may also use for primitive purposes if you wish.

One consequence of this approach is of course that the language is no longer so straightforward to implement. Another is that, although the language was designed with non-professionals in mind, it turns out to be of interest to professionals too: several people in our institute now use it in preference to other languages.

The sort of computers that we are aiming at are the very powerful personal computers just now appearing at the top end of the market. With such power at your disposal, you want a language that minimises *your* effort, not the computer's. It is not, and has never been, our intention to implement B on 8-bit micros. This would have been "designing for the past".

A taste of B

It is not the purpose of this paper to give a complete description of B, but the main features need to be described to explain the issues involved in the implementation. For further details of the language see reference [Geurts].

B is strongly typed, but variables do not have to be declared. There are two basic data types, numbers and texts, and three constructed types, compounds, lists, and tables. All types are unbounded. Thus numbers may have any magnitude, texts, lists and tables any length, within, of course, the confines of memory. There are no 'invisible' types like pointers; thus all values may be written.

Numbers are kept exact as long as possible. Thus as long as you use operations that yield exact results, there is no loss of accuracy. This includes division, so that `(1/3)*3` is *exactly* `1`. Clearly however, operations such as square root cannot in general return an exact answer, and in such cases approximate numbers are used, rounded to some length.

Texts are strings of characters. There are operations to join texts, trim them, select individual characters (themselves texts of length one), and so forth.

Compounds are the equivalent of records, for instance in Pascal, but without field names.

Lists are ordered lists of elements. The elements must be all of one type, but otherwise may be of any type. Thus you may have lists of numbers or texts, but also of compounds, of other lists, and so on. Lists may contain duplicate entries (thus they are bags or multisets, rather than sets). There are operations to insert an element in a list, remove one, determine if an element is present, and so on.

Tables are generalised arrays, mapping elements of any one type to elements of any other one type. Again there are no restrictions on the types involved. You may insert entries in a table, modify or delete them, enquire if an entry is present, and so on. There is an operator, `keys`, that when applied to a table delivers the indexes of the table as a list.

Typical commands of B are assignment

```
        PUT count+1 IN count

        PUT a,b IN b,a

        PUT {1..10} IN list
```

input/output

```
        WRITE "list=", list

        READ table EG {["John"]: 21}
```

data-structure modification

```
        INSERT 10 IN list

        DELETE table["Peter"]
```

control commands

```
        IF value in list:
            WRITE value
            REMOVE value FROM list

        WHILE answer not'in {"yes"; "no"}:
            WRITE "Please answer with yes or no"/
            READ answer RAW

        FOR value IN list:
            WRITE 2*value
```

And there are also means to define your own commands and functions.

An example

Here are the two major routines for a cross-reference generator. The first is used to save words and the list of line numbers that each word occurs at. The second is used to print out the resulting table. Note that in B indentation is used to indicate nesting, rather than using explicit BEGIN-END brackets.

```
        HOW'TO SAVE word AT line'no IN xref:
            IF word not'in keys xref:          \if not yet in the table
                PUT {} IN xref[word]           \start entry with empty list
            INSERT line'no IN xref[word]       \insert in the list

        HOW'TO OUTPUT xref:
            FOR word IN keys xref:             \treat each word in turn
                WRITE word<<20, ':'            \output it left-justified
                FOR no IN xref[word]:          \for each number in the list
                    WRITE no>>4                \output right-justified
                WRITE /                        \output a newline
```

Implementations

The original B implementation was written in 1981. It was explicitly designed as a pilot system, to explore the language rather than produce a production system, and so the priority was on speed of programming rather than speed of execution. As a result, it was produced by one person in a mere 2 months, and while it was slower than is desirable, it was still usable, and several people used it in preference to other languages. It was written in C, but there was no attempt to make it operating system independent.

The second version, just completed, is aimed at wider use, and therefore speed and portability have become an issue, though the system has also become more functional in the rewrite. It is also written in C and was produced by first modularising the pilot system, and then systematically replacing modules, so that at all times we had a running B system. It was produced in a year by a group of four.

An important decision taken very early in the rewrite was to write the code as quickly and as simply as possible, without striving for code-speed before we had any information about where to optimise. We knew that such optimisations would only be needed in a very few places, and could easily be added later, by taking profiles of the system. In the event this decision payed off handsomely: first runs of the system showed that one single routine was unexpectedly consuming 90% of the run time. This was quickly rewritten optimally, giving a great increase in speed. We deemed it worthwhile to optimise a couple of other routines which from the profiles were clearly usually being called for very particular purposes, but this was more in the field of fine tuning.

A problem that we still have is with code size: a lesson that we are only now learning is that macros in C are extremely expensive. Certain very inoffensive looking pieces of code have been found to produce great welters of code, which on investigation have been due to the use of macros. Replacing the macros with routines usually reduces the code size significantly.

Modules

A B implementation can be broadly divided into four parts: Values, Parsing, Interpretation, and Environment. B has high-level data-types, and so some effort has to go into the values module, but once it is written the full semantic power of B types is available to all modules, which is no small advantage.

Values

All values in B, with the exception of compounds, are dynamic. In the pilot implementation values were implemented as pointers to contiguous stretches of store. This made for easy programming but slow speeds if the size of a value got large, with times $\Omega(n^2)$.

Copying of values was implemented using the scheme of Hibbard, Knueven and Leverett [Hibbard] as a basis, where each value has a reference count. Copying then consists merely of copying a pointer and updating reference counts. When a count reaches zero, the associated space can be returned to the free list. On the other hand, if a value is to be modified, such as by inserting a value in a list, if its reference count is greater than one then one level of the value must first be 'uniquified' by copying it to a fresh area of store. (Only one level need be copied because for instance if the list is a list of tables, the tables need only have their reference counts updated, since they are not changed themselves.) Already unique values may be modified *in situ*.

This scheme has one outstanding feature, that the cost of copying is independent of the size of the value. Therefore there is a size of value above which reference counting becomes cheaper than ordinary copying. This critical size is rather small, and since B values easily become large, it is advantageous. Furthermore, assignments are typically the most executed sort of statement in programs, and so choosing a method that favours copying is to one's advantage.

Since all values were implemented as contiguous areas of store, inserting or deleting parts of a value involved shuffling the rest of the value up or down to make room for the new elements or to take the place of the old. While this was only on one level of the value, its cost was still proportional to the length of the value.

The new implementation still uses the reference count scheme, but instead of contiguous areas of store now uses B trees (no relation) [Krijnen] to store the values. These are a form of balanced trees, and the cost of modifying an element is only O(log n). Instead of having to copy a whole level of the value on modification, now only a sub-section of the tree needs to be copied.

Additionally two essential optimisations for B were added: representing lists such as `{1..10}` only by their upper and lower bound; this is essential for cases like

```
FOR i IN {1..1000000}:
    PROCESS i
```

and representing the result of the `keys` operator in a special way to prevent copies being taken (essentially the reference count of the table is incremented, and the result marked as a keys value); this optimisation is essential for cases like:

```
IF k IN keys t:
    WRITE t[k]
```

The final change to the values module in the new implementation was in the numeric package. B has unbounded exact rational arithmetic, but for simplicity of implementation the pilot implementation used only pairs of real numbers to represent rational numbers, using the in-built floating point facilities of the machine. This was replaced in the rewrite by a proper unbounded-arithmetic package.

Use of values by the rest of the system.

As mentioned before, it is no small advantage to have the facilities of B values at your disposal in the rest of the system. It means for example that identifiers can be implemented using texts, with no problems about limiting the length of identifiers, and more interesting, that 'environments', which are the mapping of the identifiers of variables onto their contents, can be implemented as tables mapping texts onto other B values. Furthermore, in certain places the semantics of B demand that such environments be copied (for instance to prevent side-effects when evaluating an expression); this is consequently a very cheap operation.

A notable feature of B is the so-called permanent environment. All global variables in a session survive logout, so that when you come back to your program later, all the variables have the same values as before. This obviates the need for files in B. The implementation of this was very simple. Since environments are just B tables, the permanent environment can just be written in the normal B way, using the equivalent of the `WRITE` command, but to a standard file. On re-starting, this can be read by the equivalent of the `READ` command. This has remained essentially the same in both versions, though there is now a plan to load permanent targets only when they are needed, to reduce start-up times for large permanent environments.

Parsing

In the pilot version the user's 'units' (procedures and functions) were stored on separate files in a directory. On running the B system all units in the current directory were loaded into a big buffer in main store, and represented quite literally as a stream of characters, with a special character to mark the end of each unit (this could cause problems for directories with many units, as start-up time then became rather long). Parsing then proceeded in a top-down fashion by identifying for the construct being parsed its 'skeleton' (such as `PUT ... IN ...` for an assignment; `{...}` for a list or table display) and then passing on the sub-strings for the inner constructs to be parsed. No attempt was made to produce a parse-tree or other internal representation; only the literal text form was used.

The new implementation uses the same scheme of storing units on individual files, but now only loads them on demand, i.e. when they get used the first time. Furthermore, although the same parsing scheme is used, an internal parse-tree *is* formed, and therefore only one text line at a time need be present in main store. The parse-tree has been designed to suit all purposes in the system (such as interpretation, and reconstructing the source from its internal representation) so that there need be only one canonical representation throughout the system. The representation is a fairly traditional abstract syntax tree; so, for example, a for-command representation has a node-type indicating it is a for command, and then has three sub-trees for the identifier, the expression, and the body of the for. Naturally, a parse-tree is represented using B values, by using nested compounds.

A feature of B relevant here is that expressions do not have a context-free grammar. In order to allow expressions like `sin x`, it is not always possible to decide if an expression like `f - 1` is a call of a function `f(-1)`, or the subtraction of `1` from a variable `f`. In the pilot implementation this could be ignored since expressions were parsed from context at the time of execution. We know of no case where this produced other than the effect intended, but it was possible in principal at least to construct an expression, for instance in a loop, which on the first iteration was parsed differently to the later iterations.

The new implementation takes the possibility of ambiguity into account, and the parser produces a special kind of node for expressions it cannot resolve. Just before executing a unit that contains such nodes, the ambiguity is resolved and the nodes replaced, or an error message is produced (which only happens when a unit uses a function that hasn't yet been defined).

Interpretation

In the pilot version parsing and interpretation were not separated. Thus there was a boolean variable that indicated whether the current command should be executed, and if so execution proceeded at the same time as parsing. This meant that during the execution of a WHILE for instance, the body would be parsed and reparsed each time round the loop.

The new version uses a recursive-descent interpreter that traverses the parse-tree. There is a main routine that is called with any particular node, which then splits the node into its sub-fields and uses the node-type to index a table of routines and call the relevant routine with the right number of parameters. This routine can then execute the node, calling the main routine where necessary to execute the sub-nodes.

The B Environment

B is an interactive language, and consequently allows you to modify your units during a session. The pilot version did this by writing the unit out to its file, and then calling an editor (of your choice) as a subprocess to edit the unit. On return the unit was re-read, and re-parsed immediately (without execution) so that the user could have immediate feed-back about errors. It was then straightforward to re-enter the editor to fix the errors.

The new implementation uses basically the same method. Only now there is a dedicated B editor. This is an editor that knows much about the syntax of B, and thus checks the syntax while you type, actually making impossible the standard sorts of mistyping such as unmatched brackets, and missing quotes.

Additionally it helps in reminding you with commands: when you are typing in a command, and you type a "p" as the first letter of the command, (upper or lower case), it guesses that you want a PUT command, and so displays on your screen

```
        PUT ? IN ?
```

(the underline shows where you currently are). If you did indeed want a PUT command, then you need only press the 'tab' key, and the cursor moves to the first of the two 'holes', and you can type in an expression and press tab again to move to the second hole. Similarly, the editor supplies matching brackets, so typing

```
        P<tab> (
```

gives

```
        PUT (?) IN ?
```

You get similar treatment with string quotes.

If you didn't want a PUT command, but instead wanted to invoke a unit of your own, called say PRINT, then typing an "r" after typing the "p" deletes the suggestion and gives you the following on the screen:

```
        PR?
```

and you can carry on typing the rest of the characters. In fact you can always ignore all this guessing if you want: if you type all the characters of each command, without using the 'tab' facility, you will still get the right result.

Another feature of the editor's knowledge of *B* is that it knows where there must be indentation, and so supplies it for you: if you type in the first line of a FOR command, followed by a newline, it automatically positions the cursor at the right position, for example,

```
        FOR i IN {1..10}:
            ?
```

which could have been typed as

```
F<tab>i<tab>{1..10<newline>
```

Another difference from usual editors is that the cursor, called the *focus* in the *B* system, can focus on large parts of text, such as a whole command. The focus is displayed by using some aspect of the terminal such as underlining, reverse video, or a different colour. The major advantage of such a focus is that it makes the editor command set very small. You no longer need separate keys for moving over characters, words, lines, paragraphs etc., and separate keys for deleting each category, but only keys for adjusting the size and position of the focus, and one key for deleting. It additionally alleviates such traditional problems with structured editors of changing an `IF` into a `WHILE`.

Apart from the addition of this editor to the new system, it was also made possible to edit the contents of permanent variables as well as units. Since such variables replace the traditional use of files in B, and are typically large, this facility is very welcome.

Availability of the Implementation

The Mark 1 implementation runs under Unix. It currently runs on VAX 11/780, VAX 11/750, PERQ, Philips PMDS, Bleasdale, and other 68000 systems running Unix. There is a project underway to put it on an IBM PC.

The operating system interface is localised in three files (interface with signals and interrupts, interface with the file-store, and machine parameters such as word-length) and thus transporting the system to another sufficiently large machine should cause few problems.

There is a version that runs on PDP 11/45's and similar, but it has some restrictions (such as no unbounded arithmetic) and is slower.

The system is available at nominal cost in 'tar' format (preferably) or ANSI standard labelled tape format.

The future

The B group is now engaged on the next version. This will feature further efficiency improvements, such as speeding-up sequential access to a data-structure, by far the most common case, but will mainly involve functional improvements. First of all a full B-dedicated environment will be implemented, rather the the current embryonic one. For instance, you will always be in the editor, even when typing commands to be executed immediately. Furthermore the editor will know about the semantics as well as the syntax of B, and thus many semantic checks will be performed before your unit is run. Additionally, there will be extensions such as graphics added to the system.

Conclusions

We are rapidly approaching the time when all personal computers will have the power of a VAX or greater. With power like that available, there will be less demand for languages that squeeze the last drop of power out of your processor. Our implementation project has shown that you can design a language with programmer ease as top priority and implement it with good performance.

F.P.Brookes advises in his excellent book [Brookes]: "Plan to throw one away; you will anyway". This sound advice has served us well. With relatively low initial cost we produced a prototype implementation that, while not perfect, allowed us to test our ideas, observe B programming practice, and note where optimisation was necessary and where not. With this information we could then proceed to a new version with relative ease.

References

[Brookes] F.P. Brookes, *The Mythical Man Month,* Addison Wesley, 1975.

[Geurts] L. Geurts, *An Overview of the B Programming Language, or B without Tears,* SIGPLAN Notices, December 1982.

[Hibbard] P.G. Hibbard, P. Knueven, B.W. Leverett, *A Stackless Run-time Implementation Scheme,* in Proc. 4th Int. Conf. on Design and Implementation of Algorithmic Languages, ed. R.B.K. Dewar, Courant Institute, New York, 1976.

[Krijnen] T. Krijnen and L. Meertens, *Making B Trees Work for B,* Report IW 219/83, Mathematical Centre, Amsterdam, 1983.

Behind Every Binary License is the UNIX Heritage

Brian E. Redman
Central Services Organization
Whippany, NJ 07981

Pat E. Parseghian
Princeton University
Princeton, NJ 08544

Behind Every Binary License is the UNIX[†] Heritage

Brian E. Redman

Central Services Organization
Whippany, NJ 07981

Pat E. Parseghian

Princeton University
Princeton, NJ 08544

ABSTRACT

Lately there seems to be some pessimism about the future of the UNIX system. Many who have watched its development from the earliest days feel that the system appears to grow corrupt and is no longer a model of innovation in operating system design.

UNIX was originally designed by a talented fraternity with a clear and common vision for a better computing environment. Ever since, the system has been redesigned by a diversity of people with different goals that tend to be less clear. UNIX has evolved from a simple, elegant model into one that is certainly complex and often seems convoluted. It no longer constitutes a statement of smallness, but appears to be growing unrestrictedly. It is generally accepted that the original systems provided a rich environment for a community of sophisticated computer users, as was intended. More recently it seems that UNIX is expected to be a computing panacea, and the compromises that have increased its palatability (and indeed, popularity) have reduced its effectiveness for its initial application.

One important difference between systems of the past and those that we'll see in the future is a preponderance of "binary only" applications. It is disconcerting that the "total system" may no longer be distributed, or may be available only at high costs.

The term UNIX has come to represent more than an operating system or computing environment; it represents philosophies about computing. Although the UNIX community may question the costs and motivations underlying these changes, we feel it is critical to recognize the important benefits that have been realized: UNIX and its philosophy have been spread among the computing masses and have influenced the direction of computing. The commercialization of UNIX is largely responsible for this. Other systems may have been just as revolutionary, but will never have a similar impact because they were kept private.

The following opinions are our own and are not likely to reflect those of our former employer, AT&T Bell Laboratories.

This paper is about "the cheese". Figure 1 is a reproduction of a poster that reminds us of the "proper" usage of the term UNIX. The usage dictated by these rules is a legal

[†] UNIX is a Trademark of Bell Laboratories.

interpretation of a word that symbolizes many more ideas than those enumerated.

Throughout this paper we represent the sentiments and ideas of many people. There has been a great deal written and said about the UNIX system and our opinions reflect that. We only first heard about the UNIX time-sharing system as students several years after its description was published in the Communications of the Association for Computing Machinery [1]. We did not experience its infancy, but one need not live through a period in order to appreciate it.

In part, UNIX is an operating system that can be characterized by its primitive operations. These include *read, write, open, close, fork* and *exec.* Clearly a system lacking these functions is not UNIX (though some purport to be). The right collection of system calls and their proper behaviour is necessary, but is by no means sufficient. At another level, UNIX is represented by the commands that one expects to find. We tend to distinguish between commands that are building blocks, designed to be fitted together to form new commands, and those that are subsystems or special purpose objects used in and of themselves. *Grep, sort, sed* and *tr* characterize the former. The C compiler, a news facility and some games represent the latter. By our definition a system lacking *grep* is not UNIX; likewise one without games is certainly suspect.

UNIX is also the environment in which we find it. An environment where the entire system's source is available. Such an environment is conducive to our understanding and our learning as we look at the system itself for models of good programming. This also provides us the opportunity to build on the work of others and avoid reinvention and incompatibilities. And of course source encourages us to find the causes of problems and fix them, or at least clearly specify them. The ability to move throughout the system and learn from it contributes to our better understanding of its overall workings.

Some less technical and more sociological factors also contribute to the definition of UNIX. Personalities from various universities, some government agencies and a few industrial laboratories are themselves a part of UNIX. The comments in the system's code are testimonials to these people. For example in an assembly language assist routine, uldiv.s, we find the illuminating comment, "this is the clever part". Indeed! Such style is an important part of UNIX.

Perhaps the most important aspect of this definition is the philosophy that bred the kernel, fostered the commands, and attracted the community. The philosophy of UNIX is alluded to if not defined outright in many publications by various authors (including Kernighan and Plauger [2,3], Kernighan and Ritchie [4], Kernighan and Mashey [5] and most recently in a book by Kernighan and Pike [6]). The philosophy dictates a system that is made up of small powerful functions, a system composed of the elements of programming style.

Given this definition of UNIX, how did it come about and why did it grow beyond a cult experience to legitimacy? Recently, these questions have been among those addressed by Dennis Ritchie in his Turing lecture [7] and by Kernighan and Pike in the epilogue of *The UNIX Programming Environment.* Ritchie felt that the principal technical aspect of UNIX that led to its initial popularity was that it was a simple and coherent system that pushed a few good ideas and models to the limit. As Ritchie explained, there were other circumstances that contributed to its success: it was introduced when minicomputers were first being seen as viable alternatives to large centrally administered mainframes; it was available on attractive hardware, the PDP†-11; and its development was influenced by enthusiastic and technically competent users over a relatively long period of time. Kernighan and Pike assert that "The central factor is that it was designed and built by a small number (two) of exceptionally talented people, whose sole purpose was to create an environment that would be convenient for program development." Both sources cite the fact that UNIX, because of this initial

†PDP is a Trademark of Digital Equipment Corporation

popularity, became important when its original followers entered the real world and demanded that UNIX be present.

The UNIX of the early seventies was readily available to a generation of forward-looking computer scientists. It was either virtually free (through modest licensing agreements to educational institutions) or effectively free (to other interests with significant financial resources). It required modest support in terms of hardware. It was elegantly simple and could be understood by a single person. And it provided a foundation for a large share of state of the art research because of the nature of the people who tended to seek it out.

It would be pleasant if that were the end of our story. And yet that is essentially the end of the UNIX story, because the UNIX that we have described has come and gone. As that UNIX fades away into a rosy memory a new concept of UNIX takes its place in the present. We feel they're distinct. Now we'll refer to the UNIX of the seventies as the academic UNIX, and the current commercial UNIX will be the cheese.

If we had to indict the moment when the academic value of UNIX peaked (and we're led to believe that we must) then we would say it was the time just before the VAX† was introduced. Indeed, the VAX-11/780 was a reverse Pandora's box in that it attracted evils. Thirty-two bit addressing was the harbinger of doom. Virtual memory was the crushing blow. Perhaps the constraint of the PDP-11 architecture provided the conscience that guarded UNIX.

We've had a difficult time trying to pinpoint the problems in various versions of UNIX that make us feel that something's gone wrong. We can think of various examples; the addition of uncritical system calls such as *ulimit*, the proliferation of trivial commands such as *logdir*, the overwhelming infestation of subsystems such as *lp* or *SCCS* which render a once concise manual diffuse. But these are just symptoms of an overall problem. Perhaps the best way to describe the problem is to contrast the present UNIX with its predecessor. We'll admit that people can argue convincingly that Sixth Edition UNIX wasn't perfect. But we don't support them.

Clearly the central factor underlying the brouhaha over UNIX today is money. That's a sharp contrast when we consider that had the development of academic UNIX been blessed with funds, UNIX would probably have been implemented on a PDP-10 and experienced quite a different future (had it been created at all).

We recall at the UNIX users group meeting in January, 1979 that a speaker from Western Electric talked about licensing. He answered questions, quoted policies and refused to predict the future. Even at that late date, we suppose there were relatively few commercial licensees because in June of 1980, Al Arms reported that there were twenty-four. Each time we saw the Patent Licensing manager from Western Electric in front of a UNIX gathering he reported the facts and figures. But each time, they took on more significance. Initially, UNIX licensing was a minor chore, but something the Bell System was obligated to do. Commercial licensees paid some substantial fees, yet they were insignificant in comparison to the assets of "The Telephone". However as more and more corporations expended greater and greater funds, revenues resulting from UNIX became "noticeable".

The increasing awareness of UNIX can be seen if we look at how it influenced the naming of organizations within Bell Labs. (Figure 2.) This is quite a contrast to the academic UNIX, planned, developed, integrated and supported by all of two people with a bit of help from their friends. A bulletin board message in an obscure New Jersey Ivy League university refers to the recruiting of "regiments of myrmidons". They report to managers, committees and task forces directing the development of UNIX. Their purposes are varied and their priorities are often incompatible. It seems that UNIX is attempting to fill a role as a computing panacea. The modest but powerful servant has developed a megalomaniacal personality. And naturally, it's becoming paranoid too. The clever ideas that go into UNIX are being guarded rather than shared. Its development is marked by competition rather than

†VAX is a Trademark of Digital Equipment Corporation

cooperation.

New UNIX systems are continually appearing in the marketplace. Many serve only as a disappointing reminder of what UNIX once was. The few experiences we've had with modern versions of UNIX from various suppliers have been utterly frustrating. Almost every facility we used fell apart at the slightest touch. Support was either non-existent, not helpful, or not timely. This in itself was not unsurmountable, but coupled with the lack of source code, and no user community, it was fatal. We're troubled by the practice of unbundling UNIX. Once integral parts are fast becoming "options". One has to be careful to order all the pieces or one could wind up with nothing but bits. Imagine buying a brand new Thunderbird†, and having to separately order the engine, and a wiring harness, and wheels, etc. and finding that the transmission for this model isn't quite working yet. "I'm sorry sir but you can only drive in reverse until 1st quarter 1985. However, we do have a model with overdrive in beta test." As more and more versions of UNIX appear, the value of is diluted. The community is becoming fragmented. At one time every user was a member of the same elite group. Now there are different sects, each with various orders and classes. We already suffer from the proliferation of really only two distinct UNIX versions. What will it be like when there are twenty?

Some of the main attributes of academic UNIX have paved the way to its downfall. Perhaps this is the "Peter Principle of Technology". Elegant simplicity has succumbed to complex efficiencies. Basic general designs were breeding grounds for amazing universal solutions. Portability led to incompatibility. The lakes, forests and minerals of UNIX have been polluted, cut down and strip mined.

So what? Isn't that what resources are for? Isn't that the natural course of events? We suppose it is, to some extent. But we can progress in a more rational manner. We can take advantage of the lessons of academic UNIX and still preserve its essence. It's not too late to start making the distinction between the academic UNIX that was designed to provide a fertile environment for computing research and the commercial UNIX that has grown out of it to support any number of specific applications. Perhaps academic UNIX can be preserved if it is viewed as a legitimate application itself. Ergonomic designs don't seem to take our needs into account. We don't want a "user-friendly system". We are not friendly users and neither are our colleagues. We're inconsiderate ogres without the slightest regard for the machine. We expect it to respond on command, to work endlessly and not to put up a fuss, rather like a mute slave whose only purpose is silently to obey.

As another example, the UNIX that is geared for a hostile environment (like industry) is not appropriate for our needs. Where it may be perfectly reasonable to protect users from each other by ensuring that their files are unreadable, that's nothing more than a road block to the sharing of information required in a research environment.

As UNIX itself is a product and is molded to meet the needs of a diversified consumer market, it's time to reconsider its development. It's inappropriate for it to be both an end product for the computing consumer and a base for further applications. Another UNIX must be made available to provide a sound foundation. Academic UNIX provides the model for such a system. That UNIX should be available with source, with support, and on various types of hardware. It should be considered THE certified UNIX upon which all others are based. Its development will be thoughtfully administered by a small group of dedicated monks. Perhaps that is how we can have our cake and eat it, too.

We've strayed from our abstract, but let us put it right by saying that we believe that UNIX is entering a period of high visibility and enormous growth. There's no doubt in our minds that UNIX will be the standard operating system for personal computers in this decade. Although it's clear that academic UNIX will not be appropriate for the majority of personal computing users, whatever form UNIX takes will surely be influenced by the good ideas that

†Thunderbird is a Trademark of Ford Motor Company

went into its original design. And although Jack and Jill Hacker may have no idea what UNIX was all about or what it stood for, they'll undoubtedly be subtly influenced by its underlying principles.

Epilogue: There is a subtle non-technical aspect of UNIX past. That is, as a small system, UNIX was able to be less harsh, less uniform, less antiseptic than is customary or businesslike. UNIX was a revolutionary system. It rebelled against traditional views of the responsibilities of the system and the user. UNIX was smug, irreverent, cliquish and sarcastic.

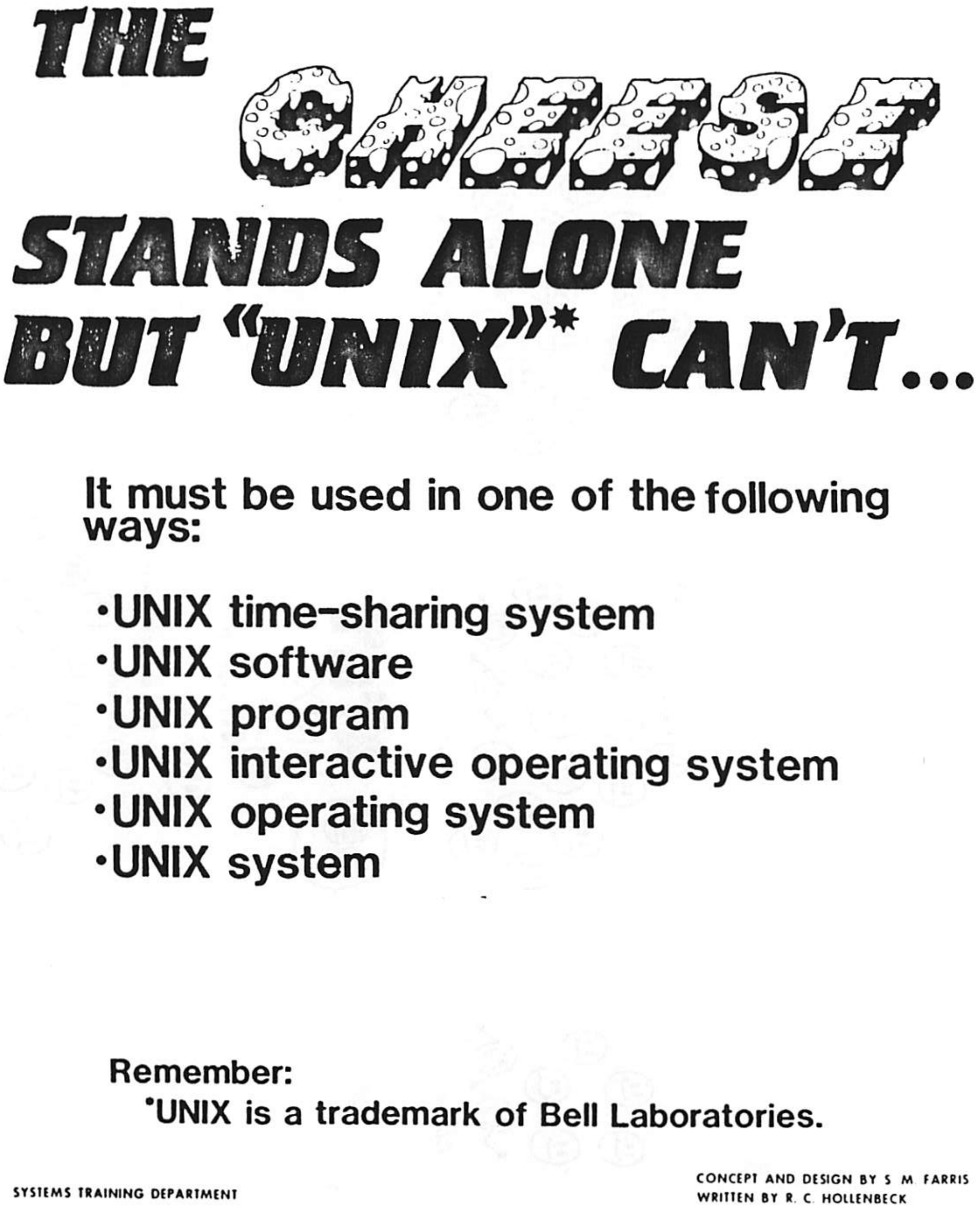
Society (computing society) has laid a burden upon UNIX. It is looking to UNIX to fulfill the promises inherent in its design and philosophy. As UNIX accepts this responsibility, it conforms to other expectations. The brash, irreverent, radical attitudes that pervade it give way to stability, clarity and uniformity. Such attributes were not necessary in the academic UNIX. In fact their absence (or lack of emphasis, to be polite) contributed to a colorful and interesting environment. We've heard that paradise is boring!

In light of IBM†'s recent announcement, we observe that UNIX has donned a suit and now fits in with its new surroundings. We hope that some of UNIX's personality will rub off on its more traditional associates. UNIX isn't going to revolutionize IBM, but its adoption by IBM is evidence of its influence.

References

1. Ritchie, D. M., and Thompson, K., "The UNIX Time-Sharing System", *Comm. ACM*, 17, 7, July 1974, pp. 365-375.
2. Kernighan, B. W. and Plauger, P. J., *Elements of Progrmming Style*, McGraw-Hill, New York, 1974.
3. Kernighan, B. W. and Plauger, P. J., *Software Tools*, Addison-Wesley, Reading, Massachusetts, 1976.
4. Kernighan, B. W. and Ritchie, D. M., *The C Programming language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
5. Kernighan, B. W., and Mashey, J. R., "The UNIX Programming Environment", *Software Practice & Experience* 9, 1, January 1979.
6. Kernighan, B. W., and Pike, R., *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, New Jersey, 1984.
7. Ritchie, D. M., Turing Lecture, 1983 ACM Annual Conference, October 24, 1983. To appear in *Comm. ACM*.

†IBM is a Trademark of International Business Machines



THE CHEESE STANDS ALONE BUT "UNIX"* CAN'T...

It must be used in one of the following ways:

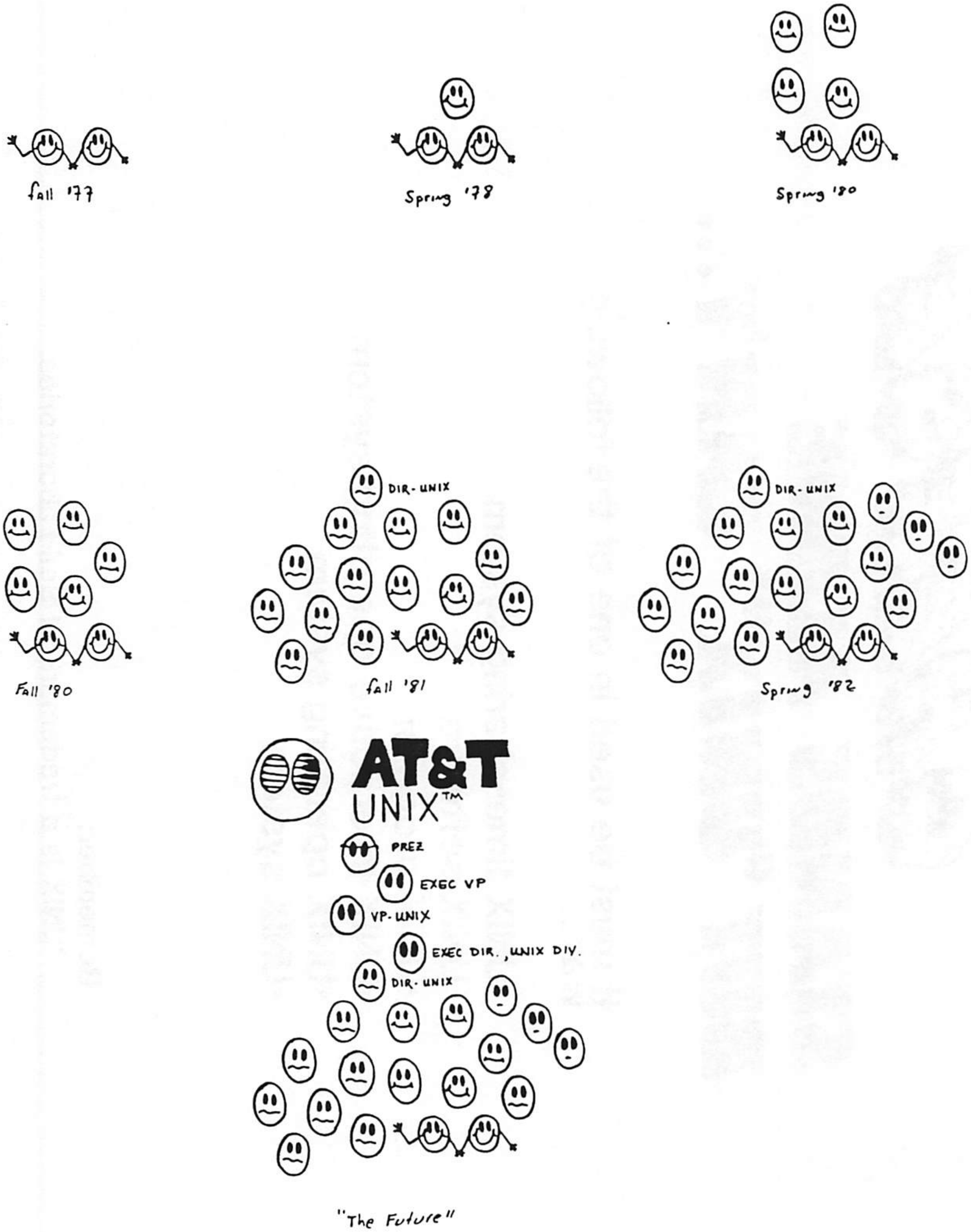
- •UNIX time-sharing system
- •UNIX software
- •UNIX program
- •UNIX interactive operating system
- •UNIX operating system
- •UNIX system

Remember:
*UNIX is a trademark of Bell Laboratories.

SYSTEMS TRAINING DEPARTMENT

CONCEPT AND DESIGN BY S M FARRIS
WRITTEN BY R. C. HOLLENBECK

Figure 1


fall '77

Spring '78

Spring '80

Fall '80

fall '81

Spring '82

"The Future"

Figure 2

Political History of UNIX

Andrew Tannenbaum
MASSCOMP
Westford, MA 01886

Political History of UNIX

Andrew Tannenbaum

MASSCOMP
Westford, MA 01886

INTRO

This isn't a talk about how Ken Thompson and Dennis Ritchie hacked their beloved and renowned PDP-7 to pieces in a tower on high.

I will not mention the extraordinary Bell Labs research environment - crafty computer science gurus who took precious time off from their highly important research efforts to help beat UNIX into shape.

I'm going to talk about some of the more earthly factors that contributed to the UNIX that we know and love today. Perhaps after hearing my story of how we got here, you'll be more able to follow in their footsteps, bringing your own products similar fame and glory, or at least you'll be better prepared to understand the results of UNIX's unusual upbringing.

There is an incredible number of people riding on the UNIX bandwagon, the size of the UNIFORUM audiences is testament to that fact.

THE CABBAGE PATCH OPERATING SYSTEM

I sometimes think of UNIX as the Cabbage Patch Operating System.

UNIX wasn't a completely new idea, it was an amalgam of good ideas.

It's certainly the latest craze, with people foaming at the mouth lining up in crazed hordes to get a peek, sometimes paying ridiculous prices for an opportunity to use the product. Of course, they say, it's worth it, there are lesser pleasures in life which are far more expensive.

Like the Cabbage Patch Kids, UNIX has existed for a while, lying around in a relatively dormant state, waiting for the market to explode.

Like Cabbage Patch Kids, every UNIX is slightly different (I'm sorry to report) but they're all similar enough to be valuable as members of the group.

Cabbage Patch Kids are the product of a company which doesn't specialize in dolls: Lately Coleco (Connecticut Leather Company) has been concentrating on the Adam computer and Colecovision home video games.

Of course, UNIX is the product of another big toy company, AT&T.

WHAT TOOK SO LONG?

You might wonder what happened forces have tugged at UNIX during the almost 15 years since UNIX was conceived.

Until recently, UNIX was never an AT&T OS product in the sense that VMS was a DEC OS product. UNIX was always, AT&T would claim, a telecommunications support tool, only because AT&T was a restricted monopoly and it was forced by legal consent decree to stay out of the computer business.

This important idea here is that AT&T wasn't allowed to compete in the computer marketplace, they weren't allowed to sell a product which would compete with IBM or DEC or any of your companies. This caused UNIX to grow up in unusual circumstances, with some benefits and some disadvantages.

In the case of most computer products, a computer company sees that the marketplace yearns for a certain product or service, and then it struggles to create that product or service before someone else does, so that it might make a killing before the rest of the market produces clones. This wasn't the case in the early days of UNIX, though it is certainly the case today.

When UNIX left the caring hands of Thompson and Ritchie, it was soon handed off to an entity called the UNIX Support Group (USG). The people who controlled the progress of USG controlled the future direction of UNIX through the 1970's.

HOW DID BELL USE UNIX?

Within AT&T, remember that UNIX was a tool, used by the Bell System projects which dealt with research and development of different parts of the telephone network - like switching, network planning, service order processing, and directory production.

UNIX was the prize in a tug-o-war between these various projects, UNIX was controlled by the telco projects with the most bucks, projects like BANCS and 5ESS, and the BTL UNIX releases sometimes had special purpose software in them with just these projects in mind. These were not the needs of the programmer, or any other common class of user, they were the specific needs of telco projects.

PROGRAMMER'S WORKBENCH

What was the Bell System going to do with UNIX?

Bell Laboratories had an enormous investment in expensive computer equipment: IBMs, UNIVAC's, Honeywells, and such. It was decided that it would be efficient to have a coherent USER interface to all these systems. What I'm trying to say here is that all the drones who punched cards for the 360's got jealous of the UNIX users with their timesharing terminals who didn't have to wait overnight for job turnaround. Today it might strike you as strange that a Teletype Model 33 user was a subject of envy: those days are gone indeed.

Anyway, one of the major UNIX forces at Bell Labs devoted itself to producing PWB, the Programmer's workbench. Programmers would learn one editor (ed) one command interpreter (sh) and be able to submit their batch jobs over rje links from one UNIX terminal. UNIX was to have COBOL syntax checkers and dump analyzers. The idea was to leave the crunching to the big batch machines, and let UNIX front end handle the humans.

Interestingly enough, I heard Bill Joy give a speech in Massachusetts where he claimed that today's UNIX workstations should be used like yesterday's terminals, and that our workstations should be networked to large mainframe CPU's which can do the supercrunching. This philosophy is not at all unlike that of PWB.

Within the Bell System, programmers used UNIX to talk to their IBM, UNIVAC, and Honeywell mainframes.

The Bell System looked at UNIX as a small machine OS. ARPA had different ideas, UNIX would be the successor to TENEX.

UNIX had no competition. Still doesn't. While this is nice, it does tend to make people who protect and defend UNIX pretty soft.

UNIX SUPPORT GROUP

Within BTL, There was a UNIX SUPPORT GROUP, but you never heard about the UNIX DEVELOPMENT GROUP. This is because no Bell System organization had the charter to DEVELOP computer operating systems. There were computer researchers, and there were people like USG who supported telco projects. No one whose charter was development. People in support positions are never given the respect that people in development positions are given.

In the beginning of USG, the name UNIX had prestige, it was a clever research toy. USG likewise had prestige, and there were clever hackers working within USG. As UNIX within BTL became more of a capitalist tool, the hackers likewise had to become capitalist tools, there was less prestige in USG, less clever work to be done, the hackers became disenchanted, and prestige and progressive development within USG disappeared.

There was no screen oriented software or OS hacks because no one had a charter to do it. The political and philosophical powers didn't want it. Sort of like the churches of the middle ages.

BELL AND BERKELEY

The UNIX development time line looked something like this:

```
research    development       maintenance     panic-explosion
                          Berkeley here.
```

BTL didn't really have a distribution policy in the early days, you got a disk with a note:

> Here's your rk05, Love, Dennis.

If UNIX crapped on your rk05, you'd write to Dennis for another. Sort of like human fsck.

There are USG UNIX User Meetings every six months, they used to always be in a big auditorium at BTL Murray Hill. For a long time there was always a carnival atmosphere at these meetings, there was always plenty of room in the auditorium, most of the congregants had lots of friends to yak with in the audience, similar to the early days of USENIX.

One of the most amusing parts of the USG UUM's was the discussion of the number of UNIX licenses that have been distributed outside the Bell System. There was always a foil which described the Bell System UNIX support policy:

> no advertising,
> no support,
> no bug fixes,
> payment in advance.

This slide was always greeted by wild applause and laughter, and, of course, there were always twice as many licensees this year as the year before. Not much to complain about.

The Bell System was fat and happy with its position in the UNIX market. Send out some tapes, rake in some bucks, not a care in the world. The USG were the Bell System's arrogant fat cats.

Then there was Berkeley. DARPA was UCB's Daddy Warbucks. UCB was a scrappy little alley cat with some street smarts to get the job done in the pinch. Where the Bell System sat on it's haunches, UCB used some crazed grad student slave labor to whip up a product that the Bell System would have taken forever to produce. The incentive just wasn't there in the Bell System.

As long as UNIX ran on PDP/11's, UNIX was never taken seriously by people with "real work" to do. No one really believed that you cold actually PORT an operating system, it just wasn't done. The PDP/11 was nice and cheap, but you couldn't run big processes on it.

EXPLOSION

When the VAX came out, you could address memory out the wazoo. Not only that, but a bunch of Bell Labbies actually PORTED UNIX to it, in *NATIVE MODE* without a gargantuan effort. The VAX had virtual memory and number crunching capability, and Berkeley had hackers interested in exploiting it.

When the VAX 11/780 emerged as a UNIX workhorse, 4.1bsd emerged as the UNIX system of choice for the VAX. Outside the Bell System, practically no one ran Bell System UNIX on their VAXes. People would buy a 32v UNIX system from WECo just for the license, they'd never even read the tape, they'd then send their license to UCB and get a copy of the latest BSD release.

Why? The UCB system was more programmer friendly. Eventually, it had paging, job control, faster I/O and higher throughput, support for many non-DEC I/O devices, reasonable though not perfect support software like screen editors and mailers, and compilers for languages like lisp and pascal. UCB users were happy because they got the toys that they wanted to play with.

BELL ATTITUDES

Eventually, smart users throughout the Bell System were running hacker-friendly 4bsd, and USG first responded with "it's not really faster, it only SEEMS faster." Well, the hackers wanted a system that seemed faster.

Why couldn't the Bell System respond? In a nutshell, BTL CS Research was too small and godly. USG was too mundane. There was nothing in between.

Yes, the Bell System does have an incredible record of innovation. They have UNIX, as well as some lesser accomplishments like the transistor and negative feedback, fundamental development of lasers and Nobel prize-winning work in radio-astronomy. They even do some telecommunications work.

Looking at it in another light, Bell Labs has more PhD's than any other organization in the world. If I remember my propaganda correctly, they have well over 7,000 PhD's, and another 20,000 engineers who can actually get the work done. While the Bell System has been a leader in innovation, they haven't done it by making efficient use of their resources, and indeed they were never under any pressure to. For this reason, a small and elite crew of hungry guerrillas like the UCB students who worked on BSD were able to have a major impact on a small piece of the Bell System's workspace.

What would have happened if there was a group of 20 hyperactive hackers developing UNIX over the years?

The BTL environment consists of mostly telephony engineers, EE's with communications backgrounds. They are ed users, 300 baud silent 700's, tek graphics scopes. Well educated, but with poor understanding of state of the art interfaces and very stubborn and arrogant. Ed and sh are just fine, thank you.

Bell was not the SAIL/ALTO/MIT/ARPA LISP/EMACS DEC10 AI environment, which is where much of the winning software from the 60's and 70's came from. Maybe not AI, but at least some good software.

Dennis Ritchie says that BTLCS was waiting for a DEC10 that never got funded. If they got it, UNIX would never have gotten off the ground. What do we owe to the Bell System administrator who turned down the DEC10?

Andrew Tannenbaum

Proposed Syntax Standard for UNIX System Commands

Kathleen Hemenway
AT&T Bell Laboratories
Murray Hill, NJ 07974

Helene Armitage
AT&T Bell Laboratories
Piscataway, NJ 08854

Proposed Syntax Standard for UNIX* System Commands

Kathleen Hemenway

AT&T Bell Laboratories, Murray Hill, NJ 07974

Helene Armitage

AT&T Bell Laboratories, Piscataway, NJ 08854

ABSTRACT

A syntax standard for user and system administrator commands is described in this paper. The objectives of the standard are to improve the quality of the user interface and to simplify development and maintenance of commands. The proposed standard is based on the syntactic features prevalent in the current command set and it is similar to, although broader in scope than, guidelines described in the *UNIX* System User's Manual* and implemented in a command line parser, *getopt*.

The proposed standard has been recommended for use by AT&T Bell Laboratories: new commands should meet the standard and use an enhanced version of *getopt*, and the standard and *getopt* should be used as a basis for regularizing the existing command set. The standard is also recommended for use by original equipment manufacturers and it will be proposed to the standards committee of /usr/group.

1. INTRODUCTION

The syntax of UNIX System commands has led some to criticize the system's user interface (e.g., [NORM81]). The syntax is inconsistent: Although most commands conform fairly well to a single syntactic model (see *intro(1)* in the *UNIX System User's Manual),* there are subtle variations among them. For example, while some commands allow more than one option to follow a single delimiter, other commands don't: *ls -l -t* may be abbreviated to *ls -lt,* but *lpstat -d -r* may not be abbreviated to *lpstat -dr*. Similarly, the significance of white space between arguments varies among commands: Some commands require white space before arguments to options, whereas others don't allow space, and in others space is optional. *Sort -o file* and *sort -ofile* are synonyms, while *cut -c list* and *cut -clist* are not. In addition to these subtle variations among commands, there are also commands that differ in major respects from the pervasive model--for example, *find* has an atypical syntax. These variations frustrate users when they are learning new commands, and the variations cause users to depend on the manual even for frequently used commands.

Inconsistencies aside, the syntax that is prevalent in the command set has been criticized. Specifically, the use of single letter options has been questioned. Some critics argue that using a single letter takes terseness too far. They argue that because a given letter typically stands for

* UNIX is a trademark of AT&T Bell Laboratories.

many different words across commands (e.g., the option *-c* stands for 'column', 'character', 'copy', etc.), it is difficult to learn and remember options. Also, the use of delimiters for options has been questioned. While a few commands use a plus sign (+) to delimit options that add features, most commands use a dash (-) to delimit all options. Since the dash is frequently conceptualized as a minus sign, this usage seems incongruent.

These issues motivated a project recently completed at AT&T Bell Laboratories. The goals of the project were to identify and evaluate shortcomings of the command syntax and to identify a syntax standard. The standard is intended to apply to new commands and to be used as a basis for retrofitting the existing command set. The standard should exert a constructive influence on the evolution of the command set by establishing a template toward which the command set will evolve. Through its implementation in a command line parser, the standard should also simplify both development and maintenance.

The proposed syntax standard and the reasoning that led to its identification are described in this paper. The standard is presented in Table 1. The processes involved in the identification of syntax rules included in the standard are described in Section 2. In Section 3 the rules are explained and the reasons for selecting the rules are described. Plans for implementing the standard and for related work are summarized in Section 4.

TABLE 1: THE PROPOSED SYNTAX STANDARD

RULE 1:	Command names must be between two and nine characters.
RULE 2:	Command names must include lower case letters and digits only.
RULE 3:	Option names must be a single character in length.
RULE 4:	All options must be delimited by "-".
RULE 5:	Options with no arguments may be grouped behind one delimiter.
RULE 6:	The first option-argument following an option must be preceded by white space.
RULE 7:	Option-arguments cannot be optional.
RULE 8:	Groups of option-arguments following an option must be separated by commas or separated by white space and quoted.
RULE 9:	All options precede operands on the command line.
RULE 10:	"--" may be used to delimit the end of the options.
RULE 11:	The order of options relative to one another should not matter.
RULE 12:	The order of operands may matter and position-related interpretations should be determined on a command-specific basis.
RULE 13:	"-" preceded and followed by white space should be used only to mean standard input.

2. *THE IDENTIFICATION PROCESS*

The proposed syntax rules were chosen on the basis of practical, technical, and user-related considerations. The match between different syntax rules and syntactic features of the existing command set was the primary practical concern: Naturally, the better the match between the new model and existing commands, the easier the transition to the new model will be. Given our commitment to upward compatibility, this was an important consideration. Technically, the command syntax should be consistent with processing accomplished by the Bourne shell, the C shell, and other popular system interface programs. Also, the syntax should be general and robust enough to facilitate argument processing across a wide variety of arguments in many commands. Finally, the syntax should present as simple a model to the user as possible, within the context of the technical and practical considerations. Although we wanted the syntax to be as simple as possible given the constraints, no attempt was made to address the special needs of naive users, because their needs can be more adequately addressed by providing an alternative, simplified interface (e.g., a menu based interface).

These concerns were addressed during a three phase study that led to the identification of the standard. A conceptual and statistical analysis of the syntactic structure of the command set was completed during the first phase. This analysis was primarily based on manual entries, and it included all the commands in Section 1 of the Release 5.0 *UNIX System User's Manual* and Section 1M of the Release 5.0 *UNIX System Administrator's Manual.* The analysis was used to determine how pervasive different syntax rules are among the commands and to assess the extent and nature of inconsistencies. It also served as a basis for defining the parts of a command. These definitions were critical to the standard since it is difficult to specify a syntax or grammar without identifying the "parts of speech". The classification of parts that was used for the standard is summarized in Figure 1.

FIGURE 1: PARTS OF A COMMAND

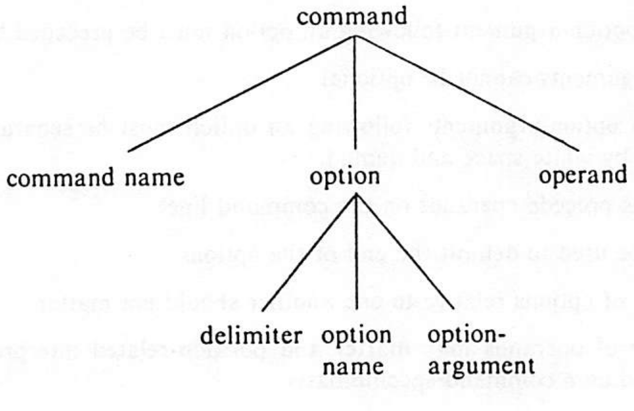


Figure 1. This figure depicts the conceptualization of parts of a command that the standard is based on. Briefly, a command is composed of a *command name* and it may have *options* and *operands.* An option is composed of a *delimiter* and an *option name* and it may have associated *option-arguments.* Operands include filenames and other parameters entered directly on the command line.

In the second phase of the study, the rules identified in the first phase were evaluated. Technical aspects of the rules were analyzed as were the human factors aspects, and the rules were compared with popular alternative rules. Syntax rules implemented at the University of Waterloo [GARD83] and rules proposed by various groups within AT&T Bell Laboratories were among the alternatives considered. Due to time and resource constraints, the human factors analyses were analytic rather than empirical--consequently, although there are many human factors arguments in this paper, experimental data are not available to substantiate them.

Finally, in the third phase of the study the strengths and weaknesses of the various rules were weighed against the impact selecting each rule would have on the current command set. A conservative criterion was used in making decisions: pervasive syntax rules were chosen over alternatives unless there was clear evidence that the benefits of making changes to the command set outweighed the costs. As it happened, in all cases where there was a single pervasive rule, it was chosen. Where the criterion of pervasiveness could not be applied, rules were selected on the basis of the technical and human factors considerations.

3. DISCUSSION OF THE PROPOSED SYNTAX RULES

Major factors that were considered in selecting the rules are summarized in this section. The impact of the structure of the existing command set on the selections is described, as are the relevant technical and human factors considerations. Strengths and weaknesses of the rules are presented and the reasons for rejecting alternative rules are summarized. Methods for compensating for the weaknesses and plans for retrofitting existing commands to meet the rules are also described.

3.1 Command Names

Rules 1 and *2* specify that command names should be between two and nine characters long and that they should include lower case letters and digits only. These constraints are consistent with existing command names (only three commands violate the rules), and they are intended to exert a conservative influence on future command names. Single character command names were excluded because they may be confused with option names and with simple editor commands. The nine character maximum length was selected to discourage the use of extremely long command names. Special characters were excluded to avoid conflicts with characters that have special meanings to the shell and upper case letters were prohibited for simplicity: As long as case does not vary in command names, users can effectively ignore it.

These constraints apply to new commands only. Due to compatibility considerations, the names of existing commands will not be changed: The few command names that violate the constraints will be left as they are.

3.2 Option Names

The use of single character option names *(Rule 3),* is pervasive in the command set: 79 per cent of the commands that have options have single character option names only. Overall, there are roughly 1400 single character names and 130 multiple character names.

Although these data largely determined the selection of *Rule 3,* before selecting it we evaluated its strengths and weaknesses relative to the strengths and weaknesses of alternative rules. Because this rule is controversial, and because its selection influenced the selection of other rules, these analyses are described below in detail.

On the positive side, *Rule 3* has several strengths. First, it is a very simple rule. Second, it minimizes the number of keystrokes and thereby minimizes (a) time to type; (b) length of the command on the command line; and (c) the frequency of spelling errors. Third, it allows bundling of options.

On the negative side, the use of single character option names has been linked to several problems. Two problems are described below: (a) contention over the use of letters; and (b) inadequate descriptive power of single letters.

1. *Contention over the use of letters.*

   A command may have more than one option that is most appropriately represented by a given letter. Typically, this is resolved by using an upper case letter for one of the options (e.g., the *awk* command uses *-f* for 'file' and *-F* for 'field separator'). This causes difficulties for monocase terminals and it causes problems when the commands are ported to monocase operating systems. It is also undesirable because case seldom has any mnemonic significance, and the user is burdened with remembering an arbitrary assignment of case to options.

2. *Inadequate descriptive power of single letters.*

   If an option can only adequately be described by a pair of words, it is difficult to select one letter for the option name. This frequently occurs when an option is most appropriately described by an action and an object. It also occurs when an option specifies the negation or suppression of an action. This leads to choices like whether to use *-s*, *-n*, or *-m* for 'no mail' (i.e., *-s* for 'suppress mail' or 'do not send mail', and *-n and -m* for 'no mail'). Because letters have been used in different ways in different commands, the meaning of the letters varies in confusing ways across commands. For example, *pr* uses *-h* to mean 'take the next argument as the header', whereas *list*, *nm*, and *prof* use *-h* to mean 'no header'.

   The use of a single letter with opposite meanings across commands is a special case of the general problem that most letters have different meanings in different commands. Although a few letters have been successfully reserved for particular uses (e.g., *-T* and *-V* mean 'output terminal type' and 'version number', respectively) most letters represent a variety of different words across commands.

Both of these problems--contention over the use of letters and the inadequate descriptive power of single letters--led us to consider alternative methods for naming options. Two alternatives to single character option names were considered: (a) multiple character option names where the whole name always must be used; and (b) multiple character option names where the minimum number of characters necessary to uniquely identify the option may be used. Both alternatives were rejected because of problems. When multiple character option names are used without truncation, naturally, typing long names is bothersome and spelling the option names is difficult for users. Although spelling an option name does not seem difficult when you consider names like "print" and "copy", it seems much more difficult when you consider that many of the names would be abbreviations. Specifically, some option names would be abbreviations of long words and others would be abbreviations of multiple word terms and phrases.

Using multi-character option names and allowing truncation reduces to the use of single letter option names in most cases, since a single letter will typically be a unique identifier for an option. Consequently, this alternative has most of the same problems as single character option names. The one problem it does not have is the contention problem: since the user may type in as many letters as are necessary to uniquely identify an option, there is no need to have unique single letters to identify options. However, there are several problems with this solution to the contention problem:

- a new option may make a previously unambiguous option ambiguous.
- to be able to identify unambiguous truncations, the user must know the entire set of legal options for a command.
- it makes bundling impractical.

In summary, neither of the two alternatives was clearly better than the use of single letter names. Consequently, we decided to endorse the de facto standard and develop ways of compensating for its limitations. Specifically, guidelines are being developed for choosing names for options. These guidelines will apply to new options only--they will not be used as a basis for changing existing single-letter options. However, selected commands with multi-character option names will be modified to accept single letter names in addition to the existing option names.

3.3 *Option Delimiters*

Rule 4 prescribes the exclusive use of dash or hyphen (-) as an option delimiter. The use of - was endorsed because it is the de facto standard: 88 per cent of the commands that have options use - exclusively as a delimiter. The use of a second, alternative delimiter was not endorsed because the costs in adding a second delimiter were considered to outweigh the benefits. Briefly, the costs are:

— another special character must be reserved.

— the user must remember when one delimiter is used rather than the other.

— the user must beware of more potential syntactic ambiguities (e.g., in addition to files named *-k*, files named *+k* become problematic).

Because of the demands already placed on special characters by the shell, we did not consider reserving a new character. Since the plus sign (+) is the only alternative delimiter that is used to any degree in the command set, it is the only alternative delimiter we seriously considered.

We decided not to endorse the use of + as a delimiter for several reasons. First, + is used in four different ways in the command set, and none of those usages met our criteria for endorsement. The criteria were: (a) there should be clear, simple, and well-defined rules for determining when + should be used; (b) the rules should be applicable to more than a few atypical commands; and (c) the use of + should complement, rather than conflict with, the pervasive use of - to delimit options. The second reason + was not endorsed was: the different uses of + in the existing command set are likely to remain for historical reasons,[1] and consequently, endorsing a single use of + in new commands could be confusing. Finally, + was not endorsed because it encourages the interpretation of - as 'minus' rather than the more neutral 'hyphen' or 'dash', which is not desirable considering that many options delimited by - seem to add features.

3.4 *Bundling*

Rule 5 states that options without option-arguments may be bundled behind one delimiter. For example, *ls -lt* may be used in place of *ls -l -t*. Bundling is a convenience that is a benefit of single character option names. It was selected because the costs are small, and the convenience is clear. Also, bundling can be used in shell scripts to make the conceptual grouping of options apparent. For example, the options *t* and *v* can be bundled in the command *cpio -i -tv -B* to indicate that *t* and *v* are related (they both affect the table of contents), and that they are separate from the other options specified.

The rule also states that an option with an argument may not occur in a bundle. For example, the command *foo -klm file* is illegal, assuming *k, l,* and *m* are options and *file* is an option-argument. This is not allowed because it makes the binding between an option and its argument unclear.

3.5 *Option-Arguments*

Rule 6 requires white space before option-arguments. This rule was selected over two alternatives: (a) no white space before option-arguments; and (b) zero or more spaces before option-arguments. Although we don't have reliable data on the numbers of commands that follow *Rule 6* and each of the alternative rules, together they account for 160 of the 163 commands that have option-arguments. In the absence of reliable data, *Rule 6* was selected on the basis of technical and human factors considerations.

Relative to using no space before option-arguments, using a space has these advantages:

[1] Specifically, the uses of + in *sh, set, sort,* and *tail* are not likely to be changed. *sh* and *set* use + to turn options off, while *sort* and *tail* use + to delimit numbers specifying a positive displacement from a numeric default.

1. When option-arguments are separated from option names by white space, they are easy to parse visually. For example, it is easier to pick out the components of *-f file* than it is to pick out the components of *-ffile*. Similarly, when option-arguments are preceded by white space, options with arguments may not be confused with bundled options or incorrectly interpreted as multi-character option names. (However, they may be incorrectly interpreted as options followed by operands if there are no subsequent options.)
2. Because the shell performs operations on words (as they are defined by white space), it is important that option-arguments be treated as words by the shell. This is clear in two cases. First, it is clear when the option-argument is a null string: for example, a null argument can be specified by the string *-d ''*, but not by the string *-d''*, because trailing quotes are stripped by the shell. Second, it is clear when filename expansion is applied to the option-argument: for example, filename expansion works appropriately on the string *-f x**. It does not work appropriately on the string *-fx** (*-fx** matches strings that begin with *-fx*).[2]

Clearly, these considerations rule out the alternative of prohibiting a space before option-arguments. However, they don't rule out the alternative of allowing space or no space. We decided to require a space, rather than allowing space or no space, because it is a simpler rule. Also, since the use of space and no space are not synonymous technically, treating them as synonyms is unwise. However, to ease the transition to requiring a space in existing commands (many of which presently allow or require no space), the ability to handle both will be provided indefinitely in the commands.[3] The documentation and training will encourage users to use a space and users will be warned that we are evolving toward requiring a space. Eventually, the commands will be changed (by making a change in the command parser) to require white space before option-arguments.

According to *Rule 7*, option-arguments cannot be optional. In other words, there are only two kinds of options--options that are always used with option-arguments and options that are never used with option-arguments. *Rule 7* follows directly from *Rule 6:* Given the choice to have white space precede option-arguments (and our goal of having simple, uncomplicated rules), option-arguments cannot be optional. This was not considered to be a significant disadvantage of the choice to have white space precede option-arguments, because our study of the command set revealed little need for optional option-arguments.

Rule 8 states that when an option has more than one argument, the option-arguments must either be separated by commas as in *foo -f file1,file2* or separated by white space and quoted as in *foo -f "file1 file2"*. In effect, this rule states that arguments to an option must be included in one word passed by the shell. This rule was selected because complications occur when option-arguments are passed as separate words by the shell and the number of option-arguments following an option may vary.

3.6 Order of Arguments

Rules 9, 11, and *12* reflect de facto standards: the vast majority of the commands require all options before operands and the order of options relative to one another rarely matters, while the order of operands frequently does.

As an alternative to *Rule 9* we considered allowing options and operands to be interspersed with no meaning attached to the order of options relative to operands. This allows the user to enter options anywhere within a command (presumably, the user may remember a neglected option after he or she has entered operands). We did not select the rule because in some commands it is clearly inappropriate to allow options to follow operands (e.g., commands that take other commands as

[2] Incidentally, these considerations preclude the use of any character as a delimiter between option names and option-arguments (e.g., =).

[3] Since *getopt* presently allows space or no space, it will support this feature without modification.

operands). Also, some commands require both options and operands in a particular, atypical order for sound technical reasons (e.g., *cc*), and those commands can more easily be viewed as exceptions to *Rule 9* than as exceptions to the alternative rule. Finally, command line editing is a better solution to the problem of "neglected options".

As an alternative to *Rule 11,* we considered a rule stating that later occurrences of an option should override earlier ones. This allows the user the flexibility to change his or her mind while entering options and to contradict him or herself. However, this feature is of limited use because there is typically no way to "erase" an option by entering another option. This feature only allows the user to: (a) enter the same option with different option-arguments, and to have the later occurring arguments take precedence over the earlier ones (which may or may not be desirable); and (b) enter mutually exclusive options that have no established precedence relation, and have the later option take precedence over the earlier one. Because the feature is not generally applicable and because command line editing is a better solution to the problem, the alternative rule was rejected in favor of *Rule 11.*

The alternative to *Rule 12*-- that is, to assigning positional interpretations to operands--is to make the operands into option-arguments, identifying by the corresponding option name the role of the option-argument. This alternative was not seriously considered.

The "--" argument, specified in *Rule 10,* may be used to keep operands that begin with - from being interpreted as options. Although operands typically do not start with -, in some cases they do. For example, it is not uncommon to *grep* for a pattern that begins with -. Also, filenames may begin with -, although this is discouraged and there are other ways around the problem (e.g., *./-k* may be used instead of *-k).*

3.7 Standard Input

Since many commands need a placeholder for standard input, there should be a convention that is used consistently across commands. *Rule 13* endorses the use of - as a pseudonym for standard input. A placeholder should be necessary only when input to a command is optional and when input is coming from more than one place.

4. GENERAL DISCUSSION

There is little that's new about the proposed standard. It is consistent with the syntax pervasive in the command set and it is largely consistent with guidelines developed in 1978 and implemented in *getopt* (see *intro(1), getopt(1),* and *getopt(3)* in the *UNIX System User's Manual).* After analyzing the problem, we decided that a conservative solution was the only sensible one: Given the software investment all users have in the current command set and AT&T's commitment to upward compatibility, practical impacts must be kept to a minimum. The proposed standard accomplishes that. At the same time, if the standard is successful it will have a positive influence on the evolution of the command set. By ensuring consistency among new commands, it will put an end to the proliferation of variation among commands. Also, by providing a model toward which existing commands can evolve, it will partially correct the existing variability.

The UNIX System Development Laboratory is moving ahead with the standardization effort internally and presenting the proposal for adoption in the larger UNIX System community. It is recommended that new commands conform to the standard and use an enhanced (smaller and faster) version of *getopt*. Mechanisms are being identified for enforcing the standard internally, and for handling "escapes" (cases where good judgment indicates that one or another rule should be broken). Plans for retrofitting the existing command set are also being identified and guidelines are being developed for designing new commands and for enhancing existing commands. These guidelines will address the problems of single character option names by establishing criteria for the selection of option names. A new standard is also being established for manual entries--the new standard will correct format problems with the current manual entries and it will ensure consistency across entries.

Acknowledgments

We are indebted to Jerry Vogel and Aaron Cohen: after evaluating the problem, we arrived at a solution very similar to the one they championed five years ago. We are also indebted to Mike Bianchi and Larry Wehr for bringing their considerable technical skills to bear on the issues addressed by the standard, and to Brian Keene for collecting data about command options. Finally, we are grateful to countless other people whose ideas helped shape the proposed solution.

REFERENCES

NORM81 Norman, D.A. The trouble with UNIX. *Datamation,* 1981, *27:* 12, 143-150.

GARD83 Gardner, J. Notes on command syntax. Paper posted to the net.cog-eng newsgroup. September 7, 1983.

MPS: A UNIX-Based Microcomputer Message Switching System

T. Scott Pyne, Joseph S.D. Yao
Hadron, Inc.
Information Technology Division
1945 Gallows Road, Suite 620
Vienna; VA 22180

MPS: A Unix(tm)-Based Microcomputer Message Switching System

T. Scott Pyne
Joseph S. D. Yao

Hadron, Inc.
Information Technology Division
1945 Gallows Road
Vienna VA 22180
(703) 790-1840

1. Introduction

This paper describes the authors' experiences as leaders of an effort currently underway at Hadron, Inc., in cooperation with Motorola, Inc., to develop a Unix(tm)-based message switching system for use in the law enforcement environment. In addition to this introduction, the paper consists of a description of the development and operational environments and the constraints they imposed on the system, a discussion of the issues that were faced when designing the system, a description of some unique mechanisms developed to ease implementation of the system's software, and some conclusions about the suitability of microcomputers and Unix for [pseudo-]real-time systems.

The MPS system was conceived as an inexpensive alternative to existing law enforcement message-switching systems. These systems typically connect mobile (in-vehicle) display terminals to state- and federal-level mainframes (such as the National Crime Information Center) for the purpose of inquiring for stolen vehicles, wanted persons, etc. Existing systems are mostly minicomputer-based and cost upwards of $200,000. The goal of Hadron and Motorola was to develop a replacement for these systems which was hosted on a microcomputer and had an entry cost less than $75,000. In addition, existing systems were written in assembly language and are thus non-portable and hard to maintain. A secondary goal was to develop the new system in C under Unix to preserve portability and enhance maintainability.

2. Environment

The MPS software package is being developed for a specific hardware and software runtime environment, which imposes significant constraints on its design. Since the target system was not available when the project was initiated, we found it necessary to use an alternate system for development. External constraints forced the choice of a development system which was less capable than the runtime target, making it necessary to accommodate the development system's idiosyncrasies to permit testing and initial system integration without restricting our ability to make maximum use of the target system's capabilities at a later time.

2.1. Hardware

The MPS package is targeted for either a Durango Poppy II or Poppy IIE microcomputer. These systems use an Intel 80286 microprocessor chip as their CPU, with an Intel 80186 as a slave processor to perform I/O. The Poppy II includes 640 Kilobytes (Kb) of memory, while the Poppy IIE comes with slightly over a Megabyte (Mb). Disk storage consists of up to 3 5.25" Winchester disk drives and 1 5.25" floppy disk drive. Maximum storage is 60 Mb (20 Mb each) on the Winchesters and 800 Kb on the floppy. The system can support up to 13 serial devices, and includes a high-speed synchronous port and a parallel interface port.

The development system in use for this project is an Altos 586-10 microcomputer. This machine uses a 10 Mhz Intel 8086 CPU, with Altos-proprietary memory management hardware. Our development machine includes 512 Kb of memory, 10 Mb of storage on 1 5.25" Winchester disk drive, and 6 serial interface ports. This is essentially the least expensive configuration of the 586, retailing for under $8000, and appears to be one of the least expensive multiuser "Unix machines" available.

These choices of hardware imposed some rather severe constraints on the application package. Perhaps contrary to some manufacturers' advertising, neither of these systems is as capable as, for example, a reasonably-well-outfitted PDP-11/44.

The primary problem seems to be the disk interface. The ST506 standard 5.25" hard disk interface defines a maximum transfer rate of 5 million bits/sec, or about 625 Kbytes/sec. This is approximately the same transfer rate as that of the DEC RL02 disk drive. The development system has only 1 of these hard disks, and the baseline version of the runtime target likewise has but 1 drive. This implies that, in the "worst case" runtime configuration, there is an absolute maximum of 625 Kb/sec disk throughput available for all purposes: operating system overhead, image swapping and/or paging, and application program I/O. In practice, we have observed disk throughput rates closer to 200 Kb/sec.

A secondary problem with the development system's hardware is the lack of an IOP (or even DMA serial I/O). Four users simultaneously running screen-intensive programs at reasonable speeds (the Berkeley vi editor at 9600 baud, for example) while one other user is compiling represent a load sufficient to exhaust all available CPU cycles. Fortunately, this is seen to be less of a problem with the run-time target, which has a dedicated IOP.

In addition to the central computer, the system includes some unique peripheral equipment which handles communication with the mobile terminals and remote mainframes.

Communication with the mobile terminals is handled by a Motorola MODAT Communications Processor, which provides an interface between an asynchronous RS232 port (to the microcomputer) and a Motorola radio transceiver station (to the terminals). This interface uses an ASCII byte stream at either 1200 or 1800 baud, consisting of message packets according to a well-defined packet structure. Except for the requirement to assemble packets for transmission and disassemble packets upon reception, this processor imposed no major constraints.

The mobile terminals themselves impose several constraints. Their display screens are formatted as 6 lines by 40 characters, with no "cursor addressing" as such. The mobile terminals operate in a block-transmit and -receive mode: an entire message packet (one screenful) is sent by the terminal in one transmission burst, while composition and editing of messages for transmission is handled by in-terminal intelligence. Reception is similarly conducted in that a message packet is accepted as a single burst transmission and stored in a reception buffer. Received messages are not displayed on the screen until the user explicitly requests them. The mobile terminals do provide the capabilities to declare guarded and/or protected fields within a message, which ease the task of forms-based interaction with the user.

Communication with the remote mainframes is accomplished via a synchronous interface implementing the IBM 3780 point-to-point "bisync" protocol. It is intended that this interface make use of the synchronous interface port on the Durango systems, along with a vendor-supplied 2780/3780 interface package.

2.2. Software

It was a contractual requirement that the application package be written in C and operate under some flavor of the Unix operating system. Earlier minicomputer-based packages of similar function had been written exclusively in assembly language and typically operated under special-purpose operating systems. Essentially, in these earlier packages, the application and the special-purpose operating system comprised one standalone application.

By developing the MPS package in C and hosting it on Unix, it was intended that portability would be maintained to a much greater degree than with this package's predecessors. The decision to develop the package in C was agreed upon by almost all involved parties. There were, however, some concerns that the package was ill-suited to run under any general-purpose operating system, especially Unix (as compared to, for example, RSX11M). The portability-vs.-performance issue was, and is, thus very much on trial in this situation.

Both the development and runtime systems operate under Xenix, Microsoft's licensed variant of the Unix operating system. The Altos provides Xenix version 2.3b, while the Durango is equipped with Xenix-286, Intel's version of Xenix 3.0 for the 80286. The use of Xenix imposes some constraints when compared to the standard Bell product.

From the development viewpoint, one major constraint imposed by this version of Xenix was the lack of a functional debugger. The delivered version of adb was only marginally useful, since it was compiled under an earlier version of Xenix with a different user structure. Since the user structure is a component of core files, the delivered adb was incapable of any postmortem actions requiring access to a core file, such as stack backtraces or examination of memory. Other utilities provided with Xenix were similarly lacking in expected features. For example, the version of the Berkeley vi editor, v2.13, supplied with Xenix does not implement the page-forward (©F) and -backward (©B) commands, and the redraw-screen command (z) works only as "z<RETURN>", not as "z." or "z-".

Xenix imposes several constraints from the operational viewpoint as well. We had intended to use the Berkeley curses screen-management package for interfacing to local CRT terminals, but upon investigation found several leftover bugs which eventually forced us to write replacements for some of the curses routines. Without these replacement routines, curses would have been useless to us. The C compiler and loader perpetuate the old PDP-11 model of the user address space, limiting a process to 64Kb of instruction space and 64 Kb of data space (which includes the stack). The Intel 8086 and successors provide several larger address space models. The Xenix kernel does in fact use (and allow user processes to use) a larger model, but programs compiled with the provided C compiler are still limited as discussed above. This constraint affected the package's performance noticeably. (It should be noted that the documentation claims that the next version of Xenix will include a C compiler which allows use of a larger address space model.)

2.3. Specifications

Insofar as the MPS package is intended to replace existing systems, it was deemed necessary for it to emulate the user interfaces of the systems it replaces. Emulation of this essentially standard user interface is further desirable because of the possibility of personnel turnover: if personnel enter an organization using the MPS package from and organization using one of the older system, similarity of the user interfaces minimizes learning time.

For these reasons, a detailed Functional Specification was prepared as a precursor to any software design. This specification described the commands and function keys available to CRT- and KDT-based users at the individual keyword level. The specification also defined layouts for the most common inquiry and message forms, in both CRT and KDT versions.

Aside from thee obvious (and intentional) constraints imposed on the design by the specification of the user interface, no major restrictions on implementation arose from the Functional Specification.

3. Implementation

The MPS system is designed as a set of cooperating processes. Each process "does only one thing and tries to do it well." However, this leads directly to the standard problems related to interprocess communications in Unix, which in turn impact the decisions of how thoroughly modularized into processes the system should be and of which parts of the system should be in which actual processes.

3.1. Overall Design

The system's primary function, of course, is simply to move messages from one peripheral to another. I/O is performed to four basic groups of devices. The mobile Keyboard Display Terminals, or KDTs, communicate through radio links to the MODAT Communications Processor (MCP) and thence via an ASCII RS-232 line to the system. These KDTs are used by field personnel to report their statuses, send messages, and make inquiries of law-enforcement databases. Up to 12 (stationary) CRTs may also be used to perform the same types of inquiry and message functions as do the KDTs, as well as administrative functions such as changing statuses for field units without KDTs. The CRTs may also be used for a "Unit Status Display", which is a realtime display of the status code associated with each KDT and the length of time that the KDT has been in a state. The system includes one or more synchronous RJE interfaces. These interfaces are used to transmit inquiries against law-enforcement databases to state- and national-level computer systems or networks. One or more printers are interfaced for use either on direct command or in those situations when a message must be rerouted from its intended destination due to excessive message size or unavailability of the destination device.

The system thus has only four major I/O subsystems, one for each class of devices. Further, the I/O subsystems for the KDT units (and associated MCP) and the CRTs are quite similar.

The same model of I/O is used for the KDTs/MCP and the CRT terminals: physical I/O is performed by an asynchronous user-level daemon, with input taking priority over output; outgoing messages are queued up by a queue server; incoming commands are processed by a transaction processor. Despite the similarity of approach, however, CRTs and KDTs differ in that CRTs are handled individually and interactively, while the nature of the MCP hardware requires handling of the KDTs in a very structured block-oriented manner. Due to this difference, it was decided that, for CRT terminals, both major portions of the I/O model would be implemented as a single process. A separate instance of this process would be created to serve each CRT on the system. For the KDTs, in contrast, it was decided that each of the model's three major functions would be implemented as a separate process, but that there would be only one instance of the resulting three-process set for each MCP. Since, in a typical system, there exists a single MCP which handles all of the KDTs, there will typically be only one I/O process set for all of the KDTs. The MCP I/O Daemon performs I/O of individual packets to and from the KDTs. Incoming traffic is sent to the KDT Transaction Processor for routing to the individual applications. Outgoing messages to the various KDTs are sorted, prioritized, and queued up for transmission to the appropriate KDTs by the MCP I/O Daemon.

For the I/O daemon portion of the RJE interface, it is thought that a vendor-supplied 2780/3780 emulator package for the target system may suffice. The RJE I/O subsystem will require a queue server to accept messages which are formatted for a given remote machine and pass them to the RJE package. Deformatting software to restructure responses from the remote systems for display by CRTs and KDTs will also be required.

The printer interface was implemented as a subset of the three-process concept from the other devices. A queue server accepts messages from other processes in the system and writes them to a spooling directory. A second process, similar to the opr or lpr off-line printing utility, reads the spool directory and outputs to the printer. A bypass mechanism is provided so that the RJE subsystem can divert very long responses from remote hosts directly to files in the spool directory instead of passing them through the IPC mechanism to the print queue server.

Transactions arrive from the KDTs and the CRTs as control codes, command lines, and forms. The appropriate command processor performs the device-dependent interpretation and passes the request along to one of several applications processors. The forms manager, one of these applications processors, keeps copies of forms for inquiries and other commands. When a command or control code is given with no arguments, a form is sent back to that terminal so that arguments may be filled in. (The returning form looks remarkably like a completed command line, which eases the task of input analysis substantially.) Other applications processors include a formatter for inquiries to be passed to the remote host, a unit status server, and (for CRTs only) various other administrative functions.

At the heart of communications is the message switching module. This is a software version of the computer bus. Different modules can attach to it and send messages to one another without having to let the message-transfer modules in on the high-level interpretations of the messages. The message switcher keeps a dynamic list of aliases and groups so that it knows, for instance, that to send a message to any KDT it must send it through the KDT Queue Server and that to send a message to a named group it must send a copy to each member of the group. This module also manages the shared memory resource to avoid both inter-process conflicts and storage overflow. The switch is instrumented so that it can keep a log of message passing if necessary.

The unit-status module, mentioned earlier, is a major portion of the system. It is used to monitor the statuses of field units as an aid in choosing units to respond to calls and in identifying units which may be encountering problems or emergency situations. The module is implemented as a server process and a display module, with interface code in the CRT command processor and KDT transaction processor modules.

The status server process maintains a list of units with associated status information. This list is updated in response to status change requests from the field units. For each unit, the status server also maintains a list of processes which should be automatically notified of changes in that unit's status. Thus, if a user at a CRT has requested display of the status of all field units in the "Traffic" group, any change to the status of any field unit in that group provokes a message from the status server to the command processor for that user's CRT.

The status display module maintains a multicolumn display on a CRT, showing, for each unit in the display, the unit's identification code, status, time in status (i.e., the elapsed time since that unit changed into that status), and a comment field. If the field unit has an emergency situation underway, the comment field is automatically overwritten with a special "Emergency In Progress" notification. Each possible status may also have associated with it a time limit (for example, 45 minutes for the status "At Scene of Call"). If a unit has exceeded the time limit for its status, this fact is noted in any status displays that include that unit, as an aid to identifying situations in field personnel may require assistance.

3.2. Implementation Issues

A major issue in the design of the package was how to accomplish interprocess communication. There were 2 basic choices: via disk and via memory. The disk-based approach had the advantages of being (conceptually) only a minor extension to the existing Unix pipe mechanism, thus being more portable, and of being more reliable, since messages stored on disk are recoverable after unexpected system failures. The memory-based approach was not seen as being as reliable or portable, but had the major advantage of speed: given microcomputer hardware (as discussed under Environment) with slow disk i/o, keeping interprocess messages in memory speeds up the system substantially. It was finally decided that the package would be developed using a set of higher-level message passing functions (named, for example, sendmsg() and getmsg()) which could be, and were, implemented in both a shared-memory-based version and a disk-based version. This allowed the choice between reliability and performance to be optimized on a per-customer basis. Changeover from one version to the other required only relinking the object modules to produce new executables.

Another issue was that of the distribution of tasks among processes. Put simply, it was necessary to decide how many separate processes there should be and which processes should do what. In addition to the theoretical basis of mapping processes to/from small, well-defined sets of tasks, the decision needed to account for the addressing limitations present in Xenix (64Kb code space and 64Kb data space). Our final distribution provided for a 3-process set for communication with the field units, a 3-process set for communication with each remote host, a 2-process set for control of each printer, and 1 process for communication with each local (CRT) terminal. In addition, there are separate processes for the status server, inquiry formatter (which prepares user inquiries for transmission to NCIC), and the part of the message switch that provides for aliasing and forwarding.

This distribution evolved on a somewhat ad-hoc basis. The single-process-per-CRT approach seemed best suited to the conversational mode of use prevalent with CRTs. By contrast, the shared-resource nature of the mobile terminals with MCP interface suggested a multiple-process environment, where input from the mobile terminals was handled separately from output to them. Separate processes were deemed necessary on a practical basis as well, since the input process (transaction processor) and output process (queue server) had to have the data space to cope with multiple (up to approximately 30) mobile terminals simultaneously.

A final design issue was the mode of interaction with CRT terminals. Some members of the design team felt strongly that customers would expect block-mode terminal "interaction", owing to long previous experience in the IBM-mainframe world. Other members felt equally strongly that block-mode terminal handling had overwhelming disadvantages in terms of portability and human-engineering. The package was initially developed assuming conversational terminal handling, but the final decision has not been made. The authors are currently investigating the feasibility of block-mode equivalents to the termlib and curses libraries (with an "/etc/blockcap" file) to parameterize the idiosyncrasies of block-mode terminals (as an attempt to remove the portability problems if block mode interaction is chosen).

3.3. IPC Mechanisms

As mentioned, we had resolved the issue of IPC mechanism essentially by avoiding it. We developed both disk- and memory-based IPC mechanisms, allowing the package to be linked with either set. In this section, we describe the specific mechanisms that were used to implement the two approaches and discuss the message-passing techniques built on top of the basic IPC. Two memory-based IPC mechanisms are actually presented: the one which we started with but later abandoned and the one we subsequently devised.

3.3.1. Pseudo-ports

Ports were an early implementation of fifo files, or named pipes, developed by the Rand Corporation. They have simple, easily defined characteristics, and we were very familiar with how they worked and how the pipe code on which they were based worked. Consequently, they were chosen as the model for our implementation of disk-based IPC.

The characteristics of ports are mimicked quite closely by our pseudo-ports. The only major difference is that writes to a port block on reaching the maximum size of a pipe (typically, 4Kb). We decided that, since speed was essential, writes to our pseudo-ports would never block; it would be up to the co-operating processes to make sure that a consumer always consumed messages faster than a producer produced them. (In an earlier version, we did in fact have a writer block at 4Kb. Speed suffered.) We also decided that, to keep from having to read a size and then a block of information, all reads and writes would be of uniform length. The information passed through the port, then, was just a header that pointed to a file where the real information was stored.

The reader, upon "opening the port", initializes an internal structure and creates the port file. The writer, on the other hand, merely initializes the structure and tests for access, returning an error if it does not have read access to the port file. A read always reads from the current file pointer. A write always opens the file for appending, writes the data at the end of the file, and closes the file. If a read returns EOF, the file is truncated so that reads and writes will start again at offset 0. To strictly emulate a port's behavior, we might have wanted to do this after any successful read if tell() matched the size of the file; but for portability, one may only assume that tell() returns a code to which one may seek, and not that a file's size can be reliably determined.

We wanted the reading and writing operations to be relatively atomic. As can be seen, they are far from being so. Therefore, for each port there is a lock file whose name is uniquely generated from the port's name. "Link locks" are used, based on the fact that the link() operation is atomic, and returns error if another process has performed the same link without unlinking. Thus a read will lock before reading and unlock after testing the return (and possibly truncating the file). A write will lock before seeking to EOF and unlock after writing.

To emulate the kernel sleep() and wakeup(), it was necessary to use either timed sleep() syscalls or pause()-kill() combinations. The former is less desirable, but was used for the link-locks on the grounds of (a) rarity of occurrence and (b) relative simplicity of code. An implementation using the latter method for link-locks was proposed but never implemented. The latter method was used for read blocks, when the port file was empty.

This technique was very portable. However, with the 10Mb Winchester and ST506 interface that we had, it was also excruciatingly slow. The poor performance was especially apparent when more than one pair of processes was using it. With one pipeline of three processes, this technique was able to pass a message in about one second. This, however, was not deemed to be sufficiently fast, considering the geometric degradation that would arise with multiple users.

3.3.2. Shared Memory I

Shared-memory techniques have been available in Unix since about 1977, with the Universitat Katholieke's kernel modifications on an early Unix Users' Group distribution tape. Such kernel modifications, however, are primarily useful if one has a source license for a given system (or is willing and able to de-compile and re-compile the whole system). Binary Unix and Unix-like systems such as Unix System V and Xenix 3.0 come with varying flavors of shared memory. Our initial development system ran Xenix 2.3b, which had no provisions for memory sharing. Given the need to preserve portability and the availability of Xenix only in binary form, it was clear that no implementation of shared memory was possible.

This point is best described as having been an impasse. We were cognizant of the need for fast IPC mechanisms, but were reluctant to abandon our portability goals. One member of the development team, however, presented a partial solution. The technique was not especially portable, but was easily applied to several versions of Unix, and seemed to be a good compromise. To describe this technique, it is first necessary to briefly describe the memory management of the Intel 8086 and of the Altos 586.

The Intel 8086 uses 16-bit short addresses and 32-bit long addresses to address 20 bits' worth of logical memory. The 8086 makes use of the concept of "segments". It has four segmentation registers: CS, DS, SS, and ES -- code segment, data segment, stack segment, and "extra" segment. A 20-bit physical address is formed from a 16-bit virtual address by choosing an appropriate segmentation register, shifting that register left four bits, and adding in the virtual address. The 32-bit long addresses are simply the concatenation of a segment value and a virtual address, or "offset", and the logical address is calculated in the same way. Segments can thus start at any 16-byte boundary, and implicitly extend for 64Kb. This implies that segments can and do overlap or be co-located.

Each specific combination of segmentation registers represents a model in Intel's terminology. Xenix 2.3b on the Altos 586 only allows two models: one with all four segmentation registers equal to the same value (0x5000), similar to the PDP-11 64Kb process model, and the other resembling the PDP-11 with separate instruction and data (I & D) spaces: the DS, SS, and ES all have the same value, but the CS is allowed to differ.

The Altos adds an extra level of mapping between logical memory and physical memory. A separate Memory Management Unit (MMU) allows up to 256 different 4Kb pages to be addressed. There are 256 word-length i/o ports, at i/o locations 0x200-0x3ff, which contain the addresses of these pages in the low 8 bits and access/status bits in the high 8 bits. The MMU takes the first 8 bits of the 20-bit address as an index into this set of ports. The physical address is then calculated by taking the lower 8 bits of that port, shifting that left 12 bits, and adding it to the low-order 12 bits of the logical address. For example: assume a DS of 0x5000 and a virtual address of 0x1fc8. The logical address would be (0x5000 << 4) + 0x1fc8 == 0x51fc8. If i/o port ((int *)0x200)[0x51], or (int *)0x2a2, is equal to 0x2b, then the physical address calculated is (0x2b << 12) + 0xfc8 == 0x2bfc8. There is no provision for limiting the accessible extent of the 4Kb page, so if a process has a page, then it may access the whole page. Scatter loading is supported by maintaining a small linked list of pages, so that an allocated set of pages need not be contiguous.

Given all this, the initial memory-sharing scheme was as follows. At boot time, /etc/rc would call a program to remove the last 8 pages (32Kb) from the linked list. At the same time, a patch would be made to /dev/mem so that the (otherwise non-functional) phys() system call would replace the last 8 pages of data space in the calling user process with these common 8 pages.

A number of problems with this approach became evident, however. Not least was the auto-patch concept. Having user programs playing with the page free list and patching kernel code space is always a potential problem. We decided that, when this was really implemented, we would patch /xenix to use the system's own mmugetm() routine to allocate the memory and to dispense it via phys(). In fact, it is easier to patch /xenix than to patch /dev/mem (see below), because we can adjust the size of /xenix if necessary.

There was no protection afforded shared memory in this scheme. Any program could get into shared memory via phys(). Our solution (of sorts): "this is not a problem." It was declared that the only programs running would be ours, anyway, so the system could be considered "trusted".

Programs that did not use separate I & D space had the last 8 pages of their I space co-located with the last 8 pages of D space. It was not clear that arbitrarily messing with the D pages might not affect I pages. To solve this, we agreed to link all processes using shared-memory code with separate I & D spaces requested.

If any of these programs were to be swapped out and back in, or even if another program were scheduled, the contents of the MMU ports would change. The solution (of sorts -- found by "excavative research") was to use the same /dev/mem writing program to put the same code into a portion of mmuset() that tests for a panic that "could never happen". Several objections were immediately voiced: (a) auto-patch [as mentioned above]; (b) duplication of code copied into the kernel; (c) eliminating a test for a panic on the grounds that it "could never happen" is not "defensive programming". A possible solution to the new objections was, as above, to patch /xenix. We would then be able to leave all the mmuset() code in and call the same subroutine that phys() calls. An objection to this was that it still did not cope with a number of problems, in particular expansion of the stack and data areas into either each other or (see below) into shared memory. Our only solution here was to attempt to de-compile all of the related code until we were fairly sure we had found and patched all the problem areas.

The last 8 pages of data space also happen to contain the stack, and a shared stack is, needless to say, not a very good idea. One proposed solution was to move the 8 common pages to the middle of data space. This would, however, force an arbitrary cutoff of data space and an arbitrary stack size. Another possible solution was to put the 8 pages back at the top of D space and move the stack below it. However, this would be hard to control, would take much more code/data space from the actual program, and would invalidate argument and stack pointers.

About this time, we felt it would be a good idea to look into another idea we had developed after having relaxed the constraints on portability and protection.

3.3.3. Shared Memory II

A number of instructions, such as "in", "out", and "halt", had been disabled in user mode. One thing that had not been disabled was the ability to change the segmentation registers. This is normally not a problem, as (a) C code can't get at them, and (b) all MMU ports except those in use by the current program are not accessible in user mode. But, of the 16 sets of 16 MMU ports, 3 were designated "Reserved for user space expansion." We decided to use those ports to do precisely that.

A small routine was written in C to allocate 16 4Kb pages -- 64Kb, instead of 32Kb -- and assign them to MMU ports a0-af (0x340-0x35f). This was placed within the startup code for Xenix. To access this memory from user-level code, all one has to do is change a segmentation register to get a logical address of 0xa????. Our first attempt was to permanently assign ES the value 0xa000. It turns out that printf() uses 8086 string instructions and assumes that ES == DS. Our current version temporarily sets DS and/or ES to 0xa000 in assembler routines for manipulating shared memory, and changes them back immediately. There are block routines for moving blocks in private memory, in shared memory, and in both directions between private and shared memory. There are also routines for moving a word into or out of shared memory or within shared memory.

To place the code within Xenix was itself an interesting procedure. Our copy of Xenix was compiled and loaded with versions of 'cc', and 'ld' that use 32-bit pointers, rather than 16-bit, for jumps, calls, and returns. The segment values in the 32-bit pointers are just the first loaded value after the start of each new load module, shifted right four bits. This should have been no problem: just compile to assembler, change and 'jmp cret' to 'jmpi cret'. The two passes of the assembler, however, disagree on just where this is implemented. We finally used an include file and a slightly modified version of the assembled code.

Putting the code into Xenix was also fairly simple. Microsoft has an "x.out" format that resembles the "a.out" format, but has a header that is twice as large, and other features. We obtained the text size of /xenix and converted that into a file offset for the conceptual equivalent of "_etext" in /xenix. We then expanded the size of this text area to 65534 bytes. (We were shooting for 65536, but there is this bug in the Xenix boot program....) We then put a routine named shmem() at or about the old "_etext" location, and changed the final jump in startup() from 'jmpi cret' to 'jmpi _shmem'. This used about 80 bytes out of the 1950 we had freed up, which left plenty of room for later expansion.

Using the assembly language block- and word-copy routines, there is now 64Kb of private memory for each of data and stack, including dynamically allocated data, as well as 64Kb of shared memory.

3.3.4. Messages

With the pseudo-ports, as mentioned, all messages were kept in files and only pointers to them were passed through the ports. To make the most efficient use of shared memory, we decided to put only very large messages out to disk. All others go completely through shared memory.

At boot time, /etc/rc calls /etc/shm_setup. This program reads a "shared memory description file", which contains data structure definitions for everything that is initially defined in shared memory. This includes a "message bin" for each process that needs to read shared-memory messages, shared-memory freelists, and various items of overhead. The last thing declared is the beginning of free memory. The setup process zeroes everything below free memory, then puts all free memory on the freelists. Memory is dynamically allocated to make fixed-size message headers and linked-list message buffers. These are then linked into the msgbin structures for the appropriate processes, still in shared memory. Locking is kludged by using the file-record system call locking() and by mapping the non-existent first 64Kb of an empty "lock" file onto the 64Kb of shared memory.

4. Conclusions

Several things have become manifestly apparent to us in the course of this project. First and foremost, our development system was woefully inadequate, owing primarily to its (lack of) disk support. We would advise anyone attempting a similar project to ensure that his/her development system has 8" disks (or at least multiple 5" disks) and an i/o processor. This also applies to application hosts, to a lesser degree. Second, with the proliferation of binary Unix systems specific structured techniques for binary extensions (read: patches) become necessary. CP/M, for example, offers the Resident System Extension package. Last, portability as a goal is extremely difficult to achieve in these micro-based binary-license environments. Despite conventional wisdom, which suggests that binary system software leads to portability by removing the possibility of kernel modifications, we found that not having source when needing to make extensions (not modifications as such) was a significant problem.

The authors would like to recognize and thank their co-workers on this project, without whom much of this would remain undone. Kurt L. Reisler and Brian K. Yasaki worked on the KDT terminal interface and other utilities. Henry R. Trapnell accomplished all of the CRT terminal interfacing and the Unit Status Display, and found all the residual bugs in curses. Also participating in the project was John Dewey, whom we thank for conceiving a shared memory approach, writing the Functional Specification, and arranging for the development system.

CAVEATS:

Unix is a trademark of Bell Laboratories, or whoever they are these days.

Xenix is a trademark of Microsoft Corporation.

A UNIX-Based Color Graphics Workstation

Rex McDowell
Metheus Corporation
5510 N.E. Elam Young Parkway
Hillsboro, OR 97124

A UNIX[1] Based Color Graphics Workstation

by

Rex Mc Dowell
Metheus Corporation
5510 N.E Elam Young Parkway
Hillsboro, Oregon 97124
(503) 640-8000
decvax!tektronix!ogcvax!metheus!rex

ABSTRACT

This paper explains the color graphics implementation of the Metheus Lambda 750 Workstation, which integrates Berkeley 4.1 UNIX with high performance color graphics. The workstation supports color windows and multiple font sizes and styles.

The Lambda Workstation is designed to support our VLSI CAD Software, which needs fast interactive graphics, Berkeley UNIX, and Virtual Memory. Our UNIX runs on two 68000's; one runs UNIX, the other is an IO processor with its own real-time kernel. The graphics engine consists of another 68000, acting as a display list manager and transformation processor, and a 2901 bit slice, which does the actual drawing. The graphics display is 1024 x 768 pixels. The system can be configured with 8, 16, or 24 bit planes. With 24 bit planes, you can display any of 16 million colors.

[1] UNIX is a trademark of Bell Laboratories

OUR APPLICATION

As we developed the workstation hardware and ported the system software, our VLSI CAD group was also developing our application software. Many of the workstation design goals were made to enhance our VLSI software.

Choosing Berkeley 4.1 UNIX for our Operating System was the obvious choice. Many of our first tools, and software designers, came from Berkeley, or from Vax's running 4.1.

The VLSI software can be divided into several groups, each having its own needs. The schematic entry program is the first step in designing a chip. It requires fast interactive graphics. The schematic is a logical description of the chip. Next comes simulating the circuit, which requires graphics and computing.

After the design has been verified, the physical layout of the chip is done with the physical layout graphics editor. Physical layout requires the designer to enter transistors, by putting colored rectangles on top of each other. The RISC 1 chip has about 450,000 rectangles. To display the entire chip at the scale a designer normally works at, would take a monitor 50Kx50K pixels. The user usually looks only at small part of the chip at one time, but scrolls through the chip with the mouse, panning and zooming. Our display processor and the bit slice made this possible. They can draw colored rectangles into eight bit planes at 16 million pixels a second.

Multiple windows were needed so that a user could look at the schematic and the actual physical layout of the chip at the same time.

SYSTEM ARCHITECTURE

Figure 1 diagrams the system architecture.

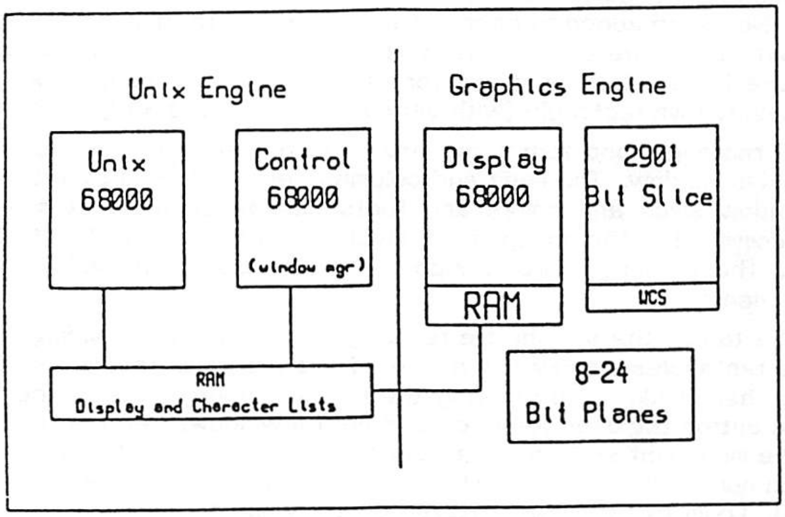


Figure 1. System Architecture

UNIX Engine

The workstation runs Berkeley 4.1 UNIX with the 4.1a network software, our own demand page virtual memory and windowing software. UNIX runs on two 68000's. The first ("UNIX 68000") runs UNIX down to the driver level. Most drivers just pass messages to the second ("Control or Interrupt 68000"). Most of the functions normally associated with UNIX drivers run under a small real-time kernel. The memory and window managers also run on the Control processor. When the UNIX processor page faults, the control 68000 gets an interrupt, validates the page for the UNIX process, and lets it continue running. The window manager gets a pointer to a character or display list, and character count, from the UNIX driver, and passes these to the display 68000, one at a time.

Graphics Engine

The display 68000 runs in its own memory, but can access memory of the other two processors. It performs the following two functions (both of which we will talk about in more detail later). The display processor handles all functions to make the color display a terminal, including things to make vi work and the multi fonts. It also is a display list processor for the bit slice. It does graphics transformations on display lists. The display processor doesn't know anything about the window management. It is just given the list of what to do, and a pointer to a structure defining the text or graphics window, current drawing color, line and solid drawing style, transformation stack and write mask.

The bit slice does the actual drawing. As it draws, it also does the clipping to window boundaries.

Shell Windows

Shell windows (screen areas to handle only text output, e.g., stdout), were implemented with only minor changes to UNIX. Each shell window is just another tty to UNIX.

Since each shell window is a tty, there is a termcap associated with each one. Several new fields were added to each termcap entry for rectangle size and location, colors, and font. There are 3 colors associated with each shell window: one for the background, one for the text, and one for the cursor or standout color. Each shell window also has its own rectangle (with variable size and location), and font style.

The shell rectangle and font sizes are used to calculate how many lines and columns fit in this window. The lines and columns fields are used by programs like vi and more. Window sizes and colors and fonts can be changed at any time with several shell commands. The changes are always reflected in the TERMCAP environment variable. These changes are only good until you close the window, or they can be made permanent.

Across the top of the screen are rectangular areas corresponding to the shell windows. The rectangles display the names of the programs that are running in the corresponding shell windows. By moving the mouse cursor to one of the rectangles and pressing a button the user can open or select a window. The first time a window is selected, the mouse interrupt routine sets the carrier on that tty, just like plugging in a terminal on normal UNIX system. This causes init to be started and init starts the login command. Login looks in */etc/ttytype* or a *.ttytype* file your home directory for the terminal type of this window, tty. It then gets the terminal description from the termcap file. Using this description, it draws your window and starts the csh. No UNIX program knows that it is talking to a color shell window.

To speed up opening a shell window, login is asleep waiting for carrier instead of init. Once carrier is given and login wakes up, it just needs to start the csh.

WINDOWS AND BIT PLANES

Color windows cause some new problems. A common way to implement black and white windows is to have all windows share one bit plane, and whenever you pop or switch window order, the window manager redraws the new top window over all the other windows, leaving all other windows showing through in the areas around the new top window. You could do this with color, but all the windows sharing bit planes would have to be the same color. If one window changes colors, then all windows will change colors; so colored window can only share planes if they have the same colors. If a program changes colors, it could be moved to its own bit planes. But very soon you will run out of bit planes.

You must have at least two colors to make a window. One color is the window background color, the other is a text color. So it takes at least two bit planes to make a window or group of windows; the 2-bit contents of each pixel is an index into a color table that gives the red, green, and blue intensities.

If you run a graphics program which needs 256 colors, then it would take 8 bit planes. If you allowed this program to run, it would change the colors of all the other windows. These new colors could even make the old windows invisible. You could just clear the screen, let the graphics program run using the entire screen and 8 bit planes, then redraw the entire screen when one of the other windows becomes top. But then what do you do with the graphic data produced by the 8-bit-plane program? 1K by 1K by 8 bit-planes is a megabyte of data. You could swap it out to disk. If you allowed several of our application programs to run at the same time, say the schematic editor, simulator and physical layout editor. You would be using 1K by 1K x 20 bit planes, or three and one half megs of data for the window manager to swap out. Or you could have the graphics program always redraw the data. However, our physical layout program occupies 12 bit-planes; it could take many seconds for it to look in its data base of hundreds of thousands of colored rectangles for which ones to draw.

Our solution is simple and very functional. Each shell window (discussed more later on) is assigned two bit planes. This gives it its own colors. If a program needs more bit planes, it makes a request to the window manager. If the planes are free, then the program can change the colormap and write into these planes without affecting any other window.

Since each window has its own planes, there is no need to redraw when you change window order. You just have to make one window appear to be on top of the other. This is done by a colormap trick. The colormap is just changed and given to the display processor. So windows are popped in the blink of an eye.

The Colormap Trick

Here is a simple example of the colormap trick. Assume we have two shell windows, each with two bit planes. There are three colors for each shell, or six colors. If you used four bit planes together, you would have sixteen possible colors. This gives us ten don't care colors. These don't care colors are always assigned to the top window.

11 XX	Bit planes for second Window
XX 11	Bit planes for Top window

00 00	Not used by either, For third window or system background
00 01	Top window color 1 (Its background color)
00 10	Top window color 2 (Its text color)
00 11	Top window color 3 (Its cursor/standout color)

01 00	Second window color 1 (Its background color)
01 01	Top window color 1 (Its background color)
01 10	Top window color 2 (Its text color)
01 11	Top window color 2 (Its cursor/standout color)
10 00	Second window color 2 (Its text color)
10 01	Top window color 1 (Its background color)
10 10	Top window color 2 (Its text color)
10 11	Top window color 2 (Its cursor/standout color)
11 00	Second window color 3 (Its cursor/standout color)
11 01	Top window color 1 (Its background color)
11 10	Top window color 2 (Its text color)
11 11	Top window color 2 (Its cursor/standout color)

OFF-SCREEN MEMORY

Figure 2 diagrams the screen format, including the off-screen memory areas.

When we started the project, the best color monitor we could get was 1024x768 pixels. All our pixel memory boards have 1024x1024 pixels on them. So before they could remove the extra memory, we used it. There are three things hidden in the off screen memory: fonts, the mouse's cursor, and the pop menu save area.

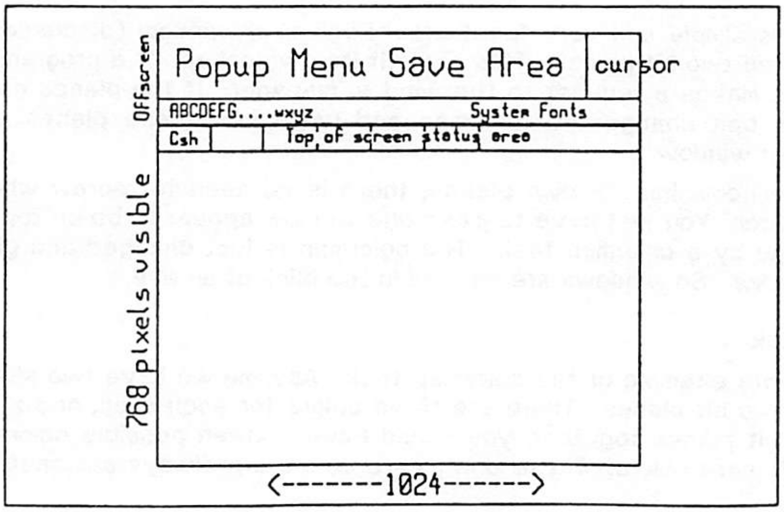


Figure 2. Screen Format

Fonts

When the system boots, it loads all the fonts from /etc/fonttable into off screen memory. A font takes only a small part of one bit plane. A character is displayed by moving it from off screen memory; the transfer takes the whole rectangle containing the character, not the individual pixels, so it is very fast.

The Graphics Cursor

The graphics cursor controlled by the mouse is a special window on the screen. The pixels for the cursor are stored in off-screen memory. The user can open a special device and draw anything in this window; this allows an application program to easily change the cursor.

Pop-Up Menus

Another special device was added for popup menus. The user opens this device, then tells the system its size and location. The system saves all the currently-displayed pixels in that location into off screen memory, then displays the pop-up menu. Using the file descriptor the user can then write to the popup menu. When the device is closed, the system restores all the pixels that were saved.

GRAPHICS PROGRAMMING

The workstation has a CORE Graphics Subroutine library. Many of the functions normally associated with CORE graphics subroutines were moved to the display processor. A new graphics driver was also added. Each tty owns at least two bit planes, and the whole screen for those bit planes. The user may break the screen into graphics windows. Each graphic window has its own size, transformation stack, drawing color, font, drawing styles, and write mask. Objects are clipped to the graphic window boundaries.

If the application program needs more colors, it may ask the system for more planes from the pool of up to 24. If the request can be satisfied, then the tty owns these planes and the colormap associated with them. The program does not know the physical locations of the bit-planes it owns, but uses logical or "virtual" numbering. For example, a program using four planes references them as planes 1-4 and colors 0-15. This way the same program could actually be run in two windows, and neither would affect the other.

Display List Objects

The application program can either send graphics opcodes such as "draw box", "draw line", or "change color" down to the display processor, or it can create an object and tell the display processor this is object 1. When the program wants to display this object, it just sends down an opcode "display object 1". These objects can be part of larger objects. Object 3 could be made up of object 2 and object 1 scaled by 1/2.

Transformations

The graphics transformations the display processor handles are scaling, rotation and translation. They are implemented by the standard matrix multiplication algorithm. These transformations are computationally intensive. By having the display processor do this, the CORE graphics library is simpler, so the UNIX processor is freed up to do other things.

TEXT HANDLING

In addition to graphics, the workstation has to handle the usual text input and output.

Text Terminal

The display processor performs all functions needed for a text terminal. Since the display processor talks to the bit slice, the text handler turned out to be a graphics program. It does everything by giving draw and move box commands to the bit slice. To print a character: "move box from off screen memory to the current character position". To clear the window: "draw box in background color". To scroll: "move a box up", then clear the bottom line, with a "draw box" command. Delete line and other special functions are also "draw box".

SUMMARY

This paper has described some of the problems involved in designing a color graphics version of the UNIX operating system, and how the solution involved advances in the hardware and software. Hardware enhancements include multiple processors, the colormap scheme and fast pixel block transfers from off-screen memory. Software design includes the new termcap, display list objects, shell windows, and other features.

User-Level Window Managers for UNIX

Robert J.K. Jacob
Naval Research Laboratory
Code 7590
Washington, DC 20375

User-Level Window Managers for UNIX

Robert J.K. Jacob

Naval Research Laboratory
Washington, D.C. 20375

ABSTRACT

Wm manages a collection of windows on a display terminal. Each window has its own shell or other interactive program, running in parallel with those in the other windows. This permits a user to conduct several interactions with the system in parallel, each in its own window. The user can move from one window to another, re-position a window, or create or delete a window at any time without losing his or her place in any of the windows. Windows can overlap or completely obscure one another; obscured windows can be "lifted" up and placed on top of the other windows.

This paper describes how such a window manager for UNIX† is implemented as a set of user processes, without modifications to the UNIX kernel. It shows how the simple, but well-chosen facilities provided by the original (Version 6) UNIX kernel are sufficient to support *wm*. In addition, subsequent versions of *wm* exploit features of the kernel introduced into newer versions of UNIX to provide faster and more sophisticated window operations, still implemented entirely at the user level.

Introduction

This paper describes the design of a display window manager for UNIX implemented entirely as a set of user processes, without modifications to the UNIX kernel. It shows how the simple facilities provided by the original (Version 6) UNIX kernel are sufficient to support such a window manager. In addition, more recent versions of the window manager exploit features of the kernel introduced into newer versions of UNIX to provide faster and more sophisticated operations in windows, still implemented entirely outside the kernel.

This window manager, *wm*, provides a UNIX user the ability to conduct several interactions in parallel, each in a different window on a text display terminal. The windows may be created, moved, and temporarily or permanently erased at any time. They may also overlap or completely obscure one another, and such hidden or partially hidden windows may be "lifted" and placed on top of the other windows as desired. Figure 1 shows a snapshot of a *wm* session in progress.

†UNIX is a trademark of Bell Laboratories.

User Interface

The notion of organizing computer data spatially was propounded and exploited by Nicholas Negroponte in the Spatial Data Management System [2, 3]. In *wm*, however, spatial cues are used only to specify a context for a dialogue. Once a window is selected, further interactions within that window make use of the power and abstraction of more conventional user interface techniques. Teitelman [8] made good use of display screen windows for a collection of parallel interactions with an INTERLISP system. More recently, several personal computers and workstations have adopted this window-oriented style of dialogue as their principal mode of interaction. Other systems similar to the present one have also been provided under UNIX [4, 7, 9].

Traditional user interfaces for computers that handle parallel processes place all inputs and outputs in one chronological stream, identifying the process associated with each, but interleaving the data. The Berkeley job control facilities for UNIX provide a first attempt at improving this situation [5].

By contrast, a window-based user interface enables a user to manage a collection of dialogues by associating a spatial location with each dialogue, in much the same way one organizes a desk. On a desk, all input papers on all topics are not (one hopes) placed on a single pile in chronological order, but rather they are divided into piles by topic. When input for a particular topic is received, the corresponding pile is located, lifted, and placed on top of other papers, and the necessary work is done on that pile. Each topic may thus be associated with and remembered in terms of a location on the desk. Recent empirical evidence showed that such a window-oriented user interface induced better user performance than a more traditional scrolled message interface in a particular situation involving several parallel interactions [6].

Wm conducts several concurrent dialogues with a UNIX user. Each takes the form of a UNIX shell, to which UNIX commands can be given and from which other interactive programs can be initiated. Each dialogue is conducted in a separate area of the screen or *window* designated by the user. At any moment, one of the windows is considered the current input window, and all keyboard inputs (except for *wm* commands themselves) are sent to the shell or program associated with that window. At any time (including in the middle of typing a command in a window), the designation of the current window may be changed and a different dialogue begun or resumed. Outputs resulting from these dialogues will appear in their appropriate windows as they are generated, regardless of which window is the current input window. Output destined for a portion of a window that is obscured by another window will appear whenever that portion of the window is uncovered. Windows can be "piled" on one another in any sequence.

Wm was originally designed for use in an intelligent terminal that could communicate with several computers simultaneously. Each dialogue with a different computer was associated with a window. The method is equally applicable to a collection of dialogues, all with the same computer but on different topics. Still, any or all of the present windows can run programs to conduct interactive dialogues with other computers (such as *telnet*).

Design for "Vanilla" UNIX

To implement a system of this sort, it is necessary for one user process to be able to manage a collection of other user processes and to mediate all of their inputs and outputs. For the inputs, it must act as a switch, directing input from the keyboard to different programs in response to user commands. For the program outputs, it must place the output of each program in its correct position on the screen.

If adequate primitives for creating and manipulating processes and for catching their inputs and outputs are provided by the operating system, a window manager can be built entirely as a user program. The original design of UNIX, with its *pipe* and *fork* mechanisms provides a user the right primitives to design such a system in an elegant fashion without kernel modifications.

Wm initiates and manages its own collection of UNIX processes, including those run in response to entered commands. Any conventional UNIX program can be used from *wm*, provided it does not make significant assumptions about the nature of its input and output devices—that is, it should treat input and output from a pipe as equivalent to input and output from a terminal or other source.

Wm runs as $2n+2$ parallel UNIX processes of four different types (where n is the number of windows in use). The division into processes is dictated by the fact that the original UNIX *read* call on an empty pipe or input device causes a process to block until input becomes available. Hence there is a separate process for each pipe or device that must be read asynchronously. Each such process contains a loop that reads from its particular pipe or device, processes the input, and then waits for more. Figure 2 shows the processes and pipes that comprise *wm*.

The **main** process reads from and waits for input from the keyboard. Input consisting of text is sent to the shell process associated with the current window and also to the **scrn** process, described below, for echoing. Input consisting of *wm* commands is interpreted by **main**, translated into one or more primitive commands plus arguments, and sent to **scrn** for execution. Simple changes in the input command language are thus localized in **main**. To change the name, input syntax, or prompt for a command, only the code in **main** need be modified. Since input commands are reduced to a somewhat more general set of primitive commands, some simple new commands may be implemented entirely in **main** as aliases for specific uses or combinations of the existing primitive commands.

The **scrn** process handles all outputs to the screen. All processes that want to affect the screen must thus place requests to do so on a common pipe. **Scrn** reads these instructions from the pipe and makes appropriate modifications to the screen. *Wm* commands that affect the screen layout, such as moving a window, are placed on this pipe by **main** and handled by **scrn**. Output text characters from the individual shell processes that belong in a window are also placed on this pipe along with a window identifier and a bit indicating whether the character should be displayed immediately or just remembered for the next time the display is refreshed. **Scrn** then compares the desired configuration of the screen to a buffer containing the actual configuration and transmits the necessary updates.†

There is a **shell** process associated with each window. This is simply the standard UNIX *sh* (or any other designated program). These **shell** processes have no direct access to the terminal, but run as captives of *wm*, connected by pipes, so that their inputs and outputs can be mediated. The input to each of these processes is a pipe from **main**, since **main** knows which window is the current input window and can place the typed input text on the pipe to the

†The update algorithm is less sophisticated than the optimization performed in the *curses* package [1], but *curses* was not available in Version 6 UNIX. This update algorithm is also somewhat easier to adapt to unusual terminals, as seen below.

corresponding **shell** process. All outputs of the **shell** processes must be sent to **scrn** to be displayed, but they must first be tagged with the name of the window in which they belong.

To do this, each window has a **shmon** process that monitors the output of the corresponding **shell** process. The output of each **shell** process is a pipe to a corresponding **shmon** process. Each time output appears on that pipe, **shmon** reads it, packages it with a header identifying its window, and then places it on the common request pipe to **scrn**.

Wm comprises about 1000 lines of C code—about 500 each for the **main** and **scrn** processes and less than 50 for the **shmon** process.

Remarks and Problems with "Vanilla" UNIX

Each window in *wm* emulates an individual glass teletype. Inputs appear in the bottom and scroll off the top of a window. Since the standard input and output for all programs run by *wm* are really pipes, all programs run under *wm* should treat their inputs and outputs simply as streams of characters, without distinctions between terminals and pipes. The fact that UNIX and most of its original programs permit a pipe to be substituted for a terminal input or output stream is an elegant aspect of UNIX that is crucial to *wm*. This obtains for most UNIX programs; they perform individual "building-block" functions and are thus intended to be equally usable individually from the terminal or as filters connected to other programs to perform more complex tasks. Programs that try to determine whether they have access to a real terminal may behave differently or even refuse to run with *wm*. For example, *stty* is meaningless when applied to a pipe rather than a terminal, *vi* will refuse to run from a pipe, and *csh* will not allow job control if it cannot access the terminal. (However, note that *wm* is really an alternate approach to controlling concurrent jobs.)

A very rudimentary facility for supporting a whole-screen-oriented program is provided. It creates a special temporary window, creates a *termcap* description of a "terminal" that occupies only the corresponding area of the actual screen, and then provides that description and direct access to the terminal to the screen-oriented program until the latter exits.

Since *wm* operates with the terminal in raw mode, it must provide for itself the input line editing functions normally provided by the teletype driver.

Because of the architecture of *wm*, there are no pipes that connect one window to another, hence there is no explicit facility for communication between windows. It can be achieved, however, through the file system. A program in one window can append to a file while one in another window continuously tries to read from the end of that file.

Terminal Dependencies

While the newer version of *wm* uses *curses* to perform all terminal-dependent operations in a terminal-independent fashion, terminal dependencies can be isolated fairly easily even without *curses*. All terminal-dependent code in the original *wm* is restricted to a collection of five simple procedures. They were originally written separately for each type of terminal, but have also been written in terms of the terminal-independent interface, *termcap*, for systems that have it.

The five procedures perform the following tasks:

ttyinit Performs any necessary initialization for the terminal.

ttyclose Performs any necessary closing for the terminal before *wm* exits or suspends.

ttymov Moves the terminal cursor to a given row and column.

clearscreen Clears the terminal screen.

clearline Clears from the cursor to end of the current line (not mandatory).

For each of several common terminals, the definitions of these procedures comprise about 15 lines of code altogether.

This approach isolates terminal dependencies sufficiently that *wm* can also be adapted for use on graphic displays by replacing the above procedures and making other minor changes. Such a version of *wm* has been written to produce output suitable for the standard UNIX plot filters (plus some added commands for raster graphic displays) and used with a Genisco frame buffer. Windows may be in various colors and may use different fonts for their text.

Design for Version 4.2 UNIX

Berkeley Version 4.2 VAX UNIX provides new features that make it possible to improve *wm* significantly. By using pseudo-terminals instead of pipes for interprocess communication, several of the problems discussed above disappear. In addition, the synchronous input/output multiplexing feature of the new UNIX makes the former division of *wm* into processes as dictated by the blocking read unnecessary. A revised version of *wm*, then, solves many of the earlier problems and runs in a single process (plus the user's shells). It is, however, less interesting and certainly less portable than the initial version. Again, the facilities are provided entirely in user-level processes, without the need for kernel modifications.

This version of *wm* reads from the keyboard and also from the pseudo-terminals associated with each window, in a round-robin, using the multiplexed read call *(select)*. Keyboard input consisting of text is sent to the pseudo-terminal associated with the current window. The pseudo-terminal driver itself handles echoing (when enabled) and intraline editing, obviating the need for *wm* to duplicate these functions. Keyboard input consisting of *wm* commands is processed directly; text input is sent to the appropriate pseudo-terminal. Output from the pseudo-terminals is read by *wm*, interpreted in terms of the cursor control commands of a simple virtual terminal defined by *wm*, and then added to the appropriate screen window for processing by the *curses* package [1].

This version of *wm* comprises about 1000 lines of C code, all in a single process. Figure 3 shows the architecture of the program.

Remarks and Problems with Version 4.2 UNIX

Since each window is implemented with a pseudo-terminal, the fact that a program is running in a window rather than on a real terminal is transparent to most programs. Specifically, most screen editors and games may be used, and *stty* may be called to change characteristics such as echoing or line editing individually for each window. For example, note that one of the windows in Figure 1 is running *vi*, which has adjusted itself to the window size. Some programs, however, assume that their output devices are of some minimum size; they will not run well in very small windows. Also, programs that attempt to manipulate the controlling terminals of process groups will not work properly under *wm*. For this reason, *csh* cannot currently be run in the individual windows instead of *sh*.

It is generally not possible to move a window while an interactive program (other than a shell) is running in it. First, this is necessary because, whenever a window is moved, *wm* sends a shell command to change the *TERMCAP* variable for the shell in that window, to describe its new size. A more fundamental reason is that the *curses* library routines (sensibly) do not expect the terminal description to change while a program is running, and so make no provision for checking for or adapting to such changes.

Since pseudo-terminals are a system-wide resource and are usually fixed in number, the total number of windows that can be in use by all users at any one time is limited to the number of pseudo-terminals made available to *wm*.

A facility for communicating between windows is now easy to provide. Since each window uses a pseudo-terminal, any data sent to its slave pseudo-terminal will appear in the window; and pseudo-terminals are in the name space of the UNIX file system and thus available to other processes. To simplify use of this feature, when a window is created and a pseudo-terminal obtained for it, a link to the name of its slave pseudo-terminal is created in the user's current directory. Any program inside or outside *wm* can then write to or read from that file name without prearrangement.

Program Versions

These programs are written in C for use with UNIX. There are three principal versions: **wm.v6**, **wm.v7**, and **wm.v42**. The first, as described above, runs under unmodified Version 6 UNIX on a PDP-11. The code for this version was frozen and abandoned several years ago, but it is still available. **Wm.v7** runs under Version 7 UNIX, and the same code also runs on Berkeley 2.8 and also on a VAX on Berkeley 4.1 and 4.2. No changes in the source code are required between the PDP-11 and VAX, except that constants for the maximum number and size of windows are limited by the available memory on a PDP-11. This version is similar in design to **wm.v6**, which was described above, but has a number of improvements. The newest version, **wm.v42**, runs only under Berkeley 4.2 on a VAX, as described in this paper. It uses the *select* synchronous input/output multiplexing call, which is unique to 4.2, and also other features that are found in some, but not all, versions of UNIX, such as pseudo-terminals and *curses*. At this writing, this version is not yet thoroughly tested on 4.2. An intermediate version for use with Versions 2.8 or 4.1 can also be constructed by adapting some of the features of **wm.v42** to **wm.v7**. For example, the use of *curses* can certainly be adapted to 2.8; pseudo-terminals are available on some versions of 4.1; and some versions of 4.1 can also simulate a non-blocking read on a pseudo-terminal or a short time-out.

Availability

Three versions of *wm* are available to interested researchers.

wm.v6	For Version 6 UNIX.
wm.v7	For Version 7 UNIX, also runs on Berkeley 2.8, 4.1, and 4.2.
wm.v42	For Berkeley 4.2 UNIX only (but has some features than can be retrofitted to **wm.v7**).

The code can be obtained over the Arpanet by sending a request to jacob@nrl-css. The author can also be reached via uucp at ...!decvax!linus!nrl-css!jacob.

Conclusions

It is demonstrably feasible to provide a useful and efficient display window management facility in UNIX at the user level, without support from kernel modifications. Such a facility can even be provided for the original Version 6 UNIX, although some improvements are obtainable by exploiting features provided by more recent versions of UNIX.

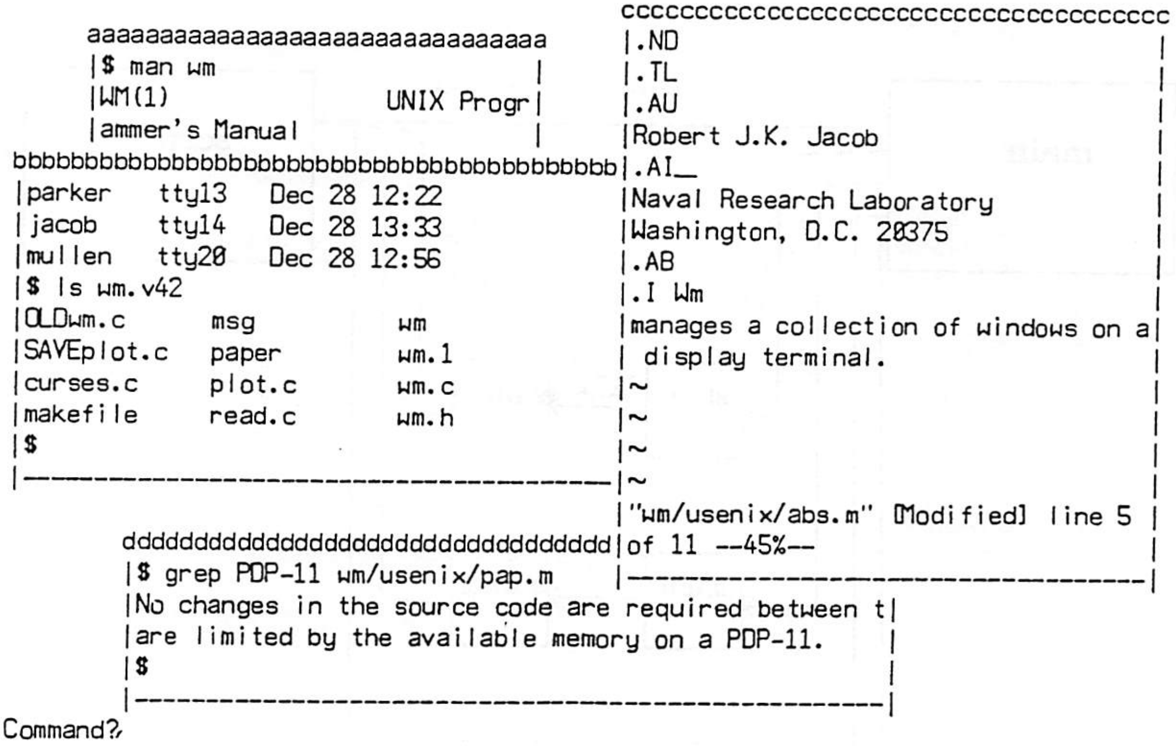
Acknowledgments

I would like to thank Mark Cornwell, Rudy Krutar, Alan Parker, and Mark Weiser for helpful discussions of this work.

References

1. K. Arnold, "Screen Updating and Cursor Movement Optimization," University of California, Berkeley (1980).
2. R. Bolt, "Spatial Data Management," Technical Report, Architecture Machine Group, Massachusetts Institute of Technology (1979).
3. C.F. Herot, R. Carling, M. Friedell, and D. Kramlich, "A Prototype Spatial Data Management System," *Computer Graphics* 14(3) pp. 63-70 (1980).
4. M. Horton, , personal communication (September 8, 1982).
5. W. Joy, "An Introduction to the C Shell," University of California, Berkeley (November 1980).
6. S. Murrel, "Computer Communication System Design Affects Group Decision Making," *Proc. Human Factors in Computer Systems Conference*, pp. 63-67 (1983).
7. R. Pike, "Graphics in Overlapping Bitmap Layers," *ACM Transactions on Graphics* 2(2)(1983).
8. W. Teitelman, "A Display Oriented Programmer's Assistant," *International Journal of Man-Machine Studies* 11 pp. 157-187 (1979).
9. M. Weiser, C. Torek, R. Trigg, and R. Wood, "The Maryland Window System," Technical Report 1271, Computer Science Department, University of Maryland (1983).

Figure 1. Snapshot of a terminal running *wm*.



```
                                         cccccccccccccccccccccccccccccccccccccc
     aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa    |.ND                                 |
     |$ man wm                      |    |.TL                                 |
     |WM(1)               UNIX Progr|    |.AU                                 |
     |ammer's Manual                |    |Robert J.K. Jacob                   |
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb|.AI_                                |
|parker   tty13  Dec 28 12:22            |Naval Research Laboratory           |
|jacob    tty14  Dec 28 13:33            |Washington, D.C. 20375              |
|mullen   tty20  Dec 28 12:56            |.AB                                 |
|$ ls wm.v42                             |.I Wm                               |
|OLDwm.c     msg          wm             |manages a collection of windows on a|
|SAVEplot.c  paper        wm.1           | display terminal.                  |
|curses.c    plot.c       wm.c           |~                                   |
|makefile    read.c       wm.h           |~                                   |
|$                                       |~                                   |
|----------------------------------------|~                                   |
                                         |"wm/usenix/abs.m" [Modified] line 5 |
       dddddddddddddddddddddddddddddddddd|of 11 --45%--                       |
       |$ grep PDP-11 wm/usenix/pap.m    |------------------------------------|
       |No changes in the source code are required between t|
       |are limited by the available memory on a PDP-11.     |
       |$                                                    |
       |-----------------------------------------------------|
Command?
```

Figure 2. Process architecture of original version of *wm*.
Boxes denote UNIX processes, lines denote unidirectional pipes.

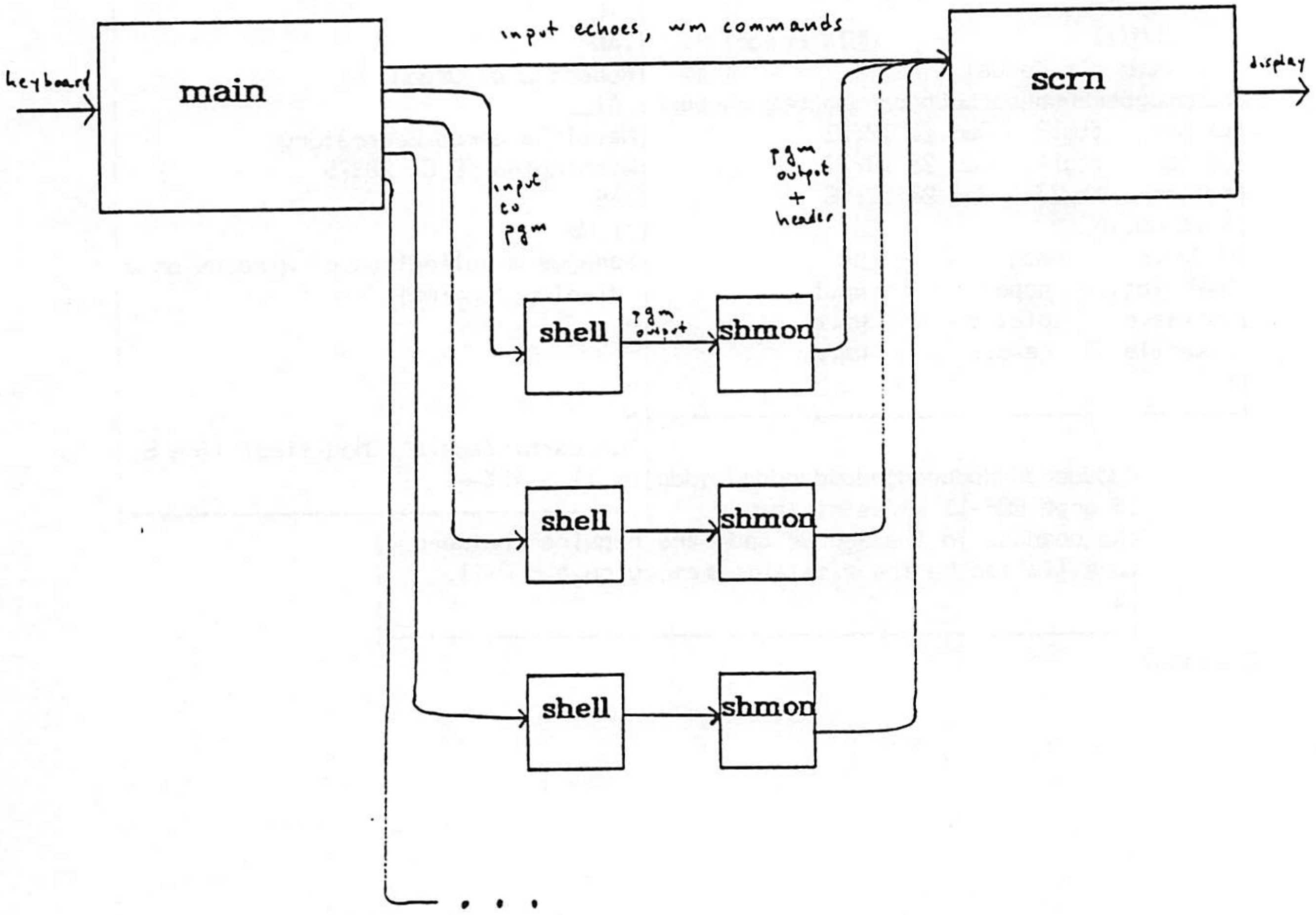
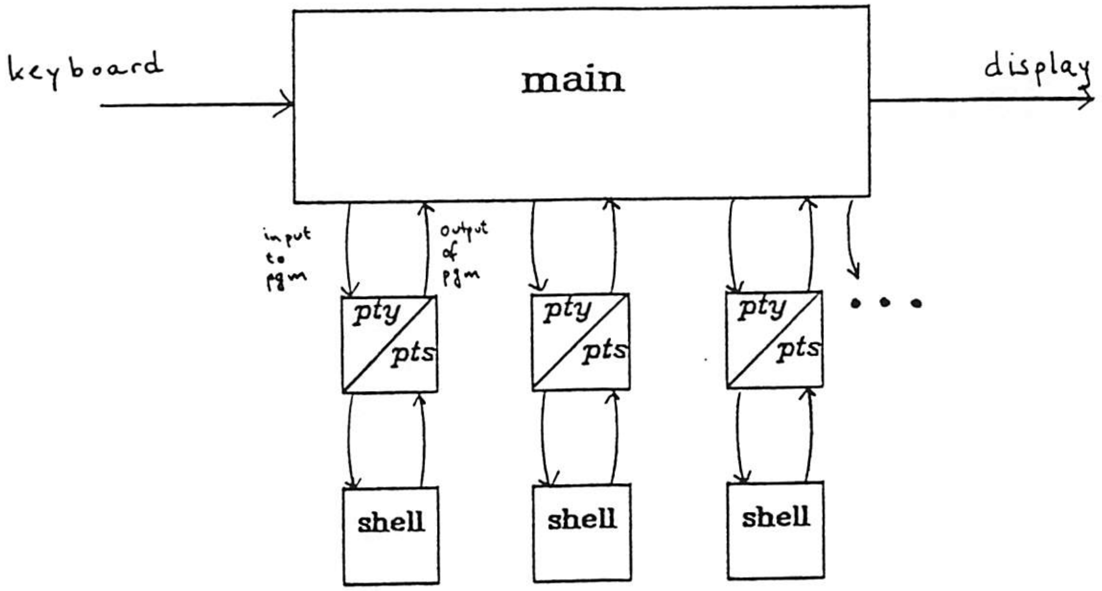


Figure 3. Architecture of *wm* for Version 4.2 UNIX.



Reloadable UNIX Device Drivers

Thomas Alborough
Creare R&D Inc.
Box 71
Hanover, NH 03784

RELOADABLE UNIX* DEVICE DRIVERS

Thomas Alborough

Creare R&D Inc.
Hanover, N.H. 03784

ABSTRACT

In order to modify a device driver with current UNIX implementations you must reboot the system. This can be a frustrating experience when only one system exists and it is being shared by several other users. Not only must the other users be asked to log off, but it also takes several minutes to stop and restart the system. This "inertia" makes it difficult to research critical operating system capabilities and idiosyncrasies, and it increases the time and effort required to develop efficient, organized and sophisticated device drivers.

Creare has developed the ability to dynamically reload the code and data portions of a device driver into a UNIX kernal while the system is running and in use by others. The capability requires a few lines of code in the device driver and a simple device loading utility. The capability should be easy to implement in old and new versions of UNIX.

Reloading a possibly flawed device driver into a system shared by others can be fatal. The changes made to a device driver during the development phase are usually incremental however, and large changes can be debugged when the system is not so busy. There are also several constraints on the changes that can be made in the device driver but in practice the method allows UNIX device drivers to be developed more quickly than would normally be possible, and with less stand-alone computer time.

1.0 INTRODUCTION

Creare R&D develops system and application level products for the Masscomp line of computers. These computers are based on the M68000 CPU and run a derivative of UNIX S/3 that includes some of the Berkley enhancements.

*UNIX Is a Bell Labs Trademark

Creare offers networking, communication and graphics products that include device drivers as part of the product. The device drivers range in complexity from ones that handle real I/O devices like synchronous/asynchronous line multiplexers, to drivers that provide asynchronous I/O capabilities, to device drivers that look like disk devices that are in reality files on some other computer on a network.

Creare had no in-house experience with UNIX when we began our development efforts, so it was critical that we not only develop the software products themselves, but also software tools and a solid understanding of UNIX and its I/O subsystem. The ability to reload UNIX device drivers at any time while the system was operating allowed us to research and make use of many UNIX kernal capabilities. We were able to develop several device drivers in parallel on a tight schedule while maintaining reasonable personal schedules.

The capability of supporting loadable UNIX device drivers is simple enough to be called a technique. The relevant files and commands are described in this paper, along with the operation of the device loading utility.

2.0 TECHNICAL OVERVIEW

This section describes the theory behind the loadable device driver technique.

2.1 How the UNIX Kernal is Built and Maintained as an Ordinary Disk File

Device drivers are written in C, compiled into object modules, and linked with other object modules to form a UNIX system image file. The format of the system image file is the same as any other runable image file, and the format is defined by the include file 'a.out.h(1)'. The system image file can only be "run" by the special codes that boot the entire UNIX system however. These codes read the system image file into memory and then begin executing it. The system image file that is normally booted is the file '/unix', although other system image file can be maintained on the system.

The system image file also sports a standard symbol table that defines the values of all global symbols and the location of all modules (callable routines). The value of specific symbols can be retrieved from the file by using the 'nlist(3)' routine.

2.2 How the UNIX System Image File Provides Access to the Running UNIX Kernal

The arrangement of code and data in the UNIX kernal when it is actually in memory is roughly the same as the arrangement when it was in the system image file. In other words if you locate code and data in the system image file by accessing the symbol table, you can then look over into system memory ('/dev/kmem') while the system is running and locate the equivalent code and data. The utility 'ps(1)' uses this scheme to locate UNIX kernal data structures in memory so that it can subsequently report on active user processes.

2.3 Fundamental Mechanics of Loadable Device Drivers

The loadable device driver technique works if three conditions are met:

> The device driver remains a fixed size as it is modified. This is accomplished by initially generating some padding in the device driver, and then (automatically) reducing the padding as the driver is enhanced. Because the device driver remains the same size from build to build this then guarantees that the rest of the routines and structures in the UNIX kernal remain in the same place from build to build. Thus there is no real difference between the subsequent versions as far as the rest of the UNIX kernal is concerned except for the modified device driver.
>
> The routines in the device driver do not appear to move as they are modified. This is accomplished with a jump table; the rest of the routines in the UNIX kernal call dummy routines at the front of the device driver and these dummy routines then call the real device driver routines. As the device driver is modified and the device driver routines move around inside the (fixed size) device driver, the rest of the UNIX kernal believes the device driver routines are staying in the same place from build to build.
>
> The driver code and data segments can be located by looking for known symbols in the symbol table, and can be checked by looking for known ASCII "marker" values in system memory ('/dev/kmem'). This accomplished by defining unique and known labels at the front and back of the device driver text and data segments, and by generating the markers at those labels.

If the above three conditions are met the device driver can be modified over and over again, and after the UNIX system image file is rebuilt the only difference between the new and old files is the contents of the code and data segments in the device driver. If just these portions of the UNIX system image file file are then copied to their proper place in system memory while the UNIX kernal is running, the device driver will have been "reloaded". This reloading can take place while the system is doing anything, so long as the device driver code and data are not in use.

3.0 TECHNICAL DETAILS OF CREARE'S IMPLEMENTATION

This section describes the technique as implemented by Creare on the Masscomp V1.0 UNIX-based system.

3.1 Relevant Files

This section describes the directories and files used to effect loadable device drivers.

3.1.1 Development Directories

A typical development environment consists of:

/unix - The default bootable UNIX kernal.

/usr/unixdev - The directory that holds all the UNIX driver development files.

/usr/unixdev/uts - The directory that holds the UNIX kernal makefile and the UNIX development files (see Section 3.1.2).

/usr/unixdev/xx - The directory that holds all the 'xxdriver' development files (see Section 3.1.3).

3.1.2 The UNIX Kernal Makefile and the Kernal Build Process

A makefile for the UNIX kernal builds the kernal from object modules. The most relevant line in the makefile is the one that invokes the linker utility 'ld':

```
ld -e start -o unix conf.o low.o ml.o os.o io.a xxdriver.o
```

Note that the majority of the UNIX kernal code and data comes from files containing concatenated object modules and from the object library 'io.a' which contains all the device drivers. The device driver under development ('xxdriver') is specifically cited in the command line in order to eliminate the time it takes to re-insert the module in the device driver object library 'io.a' library. This reduces development turn-around time.

3.1.3 The Device Driver Makefile and the Driver Build Process

The makefile for the loadable device driver operates as follows:

```
cc -S -o xxdriver.s xxdriver.c        # Compile the driver into an
                                        assembly language file.

cat xxdriver_pre.s xxdriver.s xxdriver_pst.s  |
   as -o xxdriver.o                    # Assemble the device driver with
                                        the prefix and postfix files.
```

```
     cp xxdriver.o ../uts/xxdriver.o      # Move the object module to
                                            the UNIX build area.
```

The prefix and the postfix files accomplish most of the technical preconditions described in Section 2.3.

3.2 Loadable Driver Code and Data Organization

The bulk of the requirements for a loadable driver (Section 2.3) are implemented by the device driver prefix and postfix files. The three main requirements are:

Pad the device driver code and data segments so they remain a fixed size from build to build.

Provide a jump table at the front of the device driver so that references to the device driver modules by UNIX kernal routine always point to entry points that do not move from build to build.

Provide a naming and checking scheme so the device driver code and data areas can be located and verified in by the system image files and system memory ('/dev/kmem').

The major highlights of the implementation are shown below:

--- Start of the prefix file xxdriver_pre.s ---

```
        .data                               # Set to the data segment.
        .globl xxdriver_data_start          # Make the symbol global.
xxdriver_data_start:                        # Note the front of the data segment.
        .asciz /xxdriver_data_start/        # Mark the front of the data segment.

        .text                               # Set to the text segment.
        .globl xxdriver_text_start          # Make the symbol global.
xxdriver_text_start:                        # Note the front of the text segment.
        .asciz /xxdriver_text_start/        # Mark the front of the text segment.


        .globl xx_open                      # Make the symbol global.
_xx_ _open:                                 # Define external entry point.
        jmp _xx_open                        # Jump to the internal entry
                                              point.

        .globl xx_close                     # Make the symbol global.
_xx_ _close:                                # Define external entry point.
        jmp _xx_close                       # Jump to the internal entry
                                              point.
        ...                               # (Repeat for the rest of the
                                            device driver entry points.)

--- Start of the device driver xxdriver.c ---


# if defined(xxloadable)                  /* If the device driver is
                                             loadable */
#    define xx_open xx_ _open             /* Re-define the open entry
                                             point */
#    define xx_close xx_ _close           /* Re-define the close entry
                                             point */
        ...                               /* (Re-define the rest of the
                                             device driver entry points) */
# endif                                   /* End if the device driver is
                                             loadable */


xx_open () ...
xx_close () ...                           /* Device driver modules ... */


--- Start of the postfix file xxdriver_pst.s ---


        .data                             # Go to the data segment.
        . = xxdriver_data_start + 2000    # Pad the data segment to a fixed size.
        .globl xxdriver_data_end          # Make the symbol global.
xxdriver_data_end:                        # Note the back of the data segment.
        .asciz /xxdriver_data_end/        # Mark the back of the data segment.

        .text                             # Go to the text segment.
        . = xxdriver_text_start + 5000    # Pad the text segment to a fixed size.
        .globl xxdriver_text_end          # Make the symbol global.
xxdriver_text_end:                        # Note the back of the text segment.
        .asciz /xxdriver_text_end/        # Mark the back of the text segment.
```

Note several points:

The front and back of the device driver code and data segments are defined by unique global names. Similar unique data values in the code and data segments "mark" that actual locations themselves.

The uniqueness and the actual values of the symbols and markers is a matter between the device driver and the device driver loading utility and up to the individual developer.

The padding is accomplished by using the assembler to advance the current location counter ('.' - dot) to a fixed offset from the front of the data and text segments.

The jump table sits at the front to the device driver and the labels it refers to are made different from the "normal" labels by conditionally compiling the device driver for loadable operation. The jump table must be at the front and in the loadable portion of the device driver.

3.3 Device Driver Loading Utility -- dvl

A simple utility 'dvl' is required to copy the device driver code and data segments from the system image file to system memory. The utility is invoked as follows:

```
# dvl -i [system image file name] [device driver prefix]
```

where:

The default for the 'system image file name' is './unix', (the file '/unix' in the default directory).

The 'device driver prefix' is a two letter prefix which is added to various symbol names inside the loading utility before it looks for symbols.

For example:

```
# dvl xx
```

loads the 'xx' driver from the file 'unix' into the running system.

The steps followed by the device driver loading utility are:

(1) Make a system image file name and unique device driver label names from the command line that invoked the 'dvl' utility.

(2) Open the system image file and '/dev/kmem'.

(3) Locate all the necessary symbols in the system image file by using the 'nlist(3)' routine. If any symbols are missing flag the error and exit.

(4) Locate the proper ASCII values in the code and data segments in both the system image file and '/dev/kmem'. This insures the device driver is exactly where we think it is in both places. If any discrepancies occur flag them and exit.

(5) Copy the code and data portions from the system image file to '/dev/kmem'. The device driver has now been reloaded.

3.4 Typical Device Driver Development Sequence

The device driver set-up scheme described in Section 3.2 is coded and built into a UNIX kernal image file. The system image file is booted and tested to see that the fundamental UNIX capabilities operate properly. The system image file is then moved from the development area to its the default location '/unix', which allows the standard UNIX utilities like 'ps' to work normally.

The device driver is then modified and a new UNIX system image file is rebuilt and left in the development area '/usr/unixdev/uts'. The device driver loading utility 'dvl' is used to copy the modified device driver code and data segments from the newly built system image file into system memory ('/dev/kmem').

4.0 CAVEATS

The obvious caveat is that the system is at risk every time a possibly flawed device driver is loaded and exercised. In practice this has not been a problem. If the developer warns the rest of the users with a broadcast message and if the system supports restartable editors so that program editing sessions are not lost when the system crashes, loadable drivers present real no problem.

Several things must be kept in mind in any implementation:

Note what data is in the unnamed data segment and what data is in "global shared psects" and initialize your device driver structures accordingly. Global system structures (such as 'u') and structures defined outside of the device driver modules but in the device driver source module are defined to the assembler as "global shared psects". Global shared psects are gathered by the linker utility 'ld' and put in areas of the UNIX kernal not loaded by this technique.

Do not add or delete global structures from build to build. If you suddenly define a new global structure or include a new system structure definition file, more global shared psects are generated, and this may shift around the rest of the executive structures. Depending on the set-up in the UNIX kernal the actual location of the device driver data segments may not change, and the verification scheme in the device driver loading utility may not notice that anything has changed.

Both the text and data segments in the UNIX kernal must be writable. Note that the device driver loading utility must write to '/dev/kmem'. Some system write protect the kernal text portions under the assumption that nobody should be writing on it. This feature must be disabled before the loader can fully reload the device driver; the possible methods of doing this are endless and left to the implementor.

Note the scheme used to locate the text and data segments in the system image file. In our UNIX implementation the text and data segments are aligned on 4K byte boundaries, however the linker utility 'ld' did not pad the text segment in the system image file to a 4K byte boundary (to save space). Thus, the text and data was locatable in memory, and the text was locatable in the system image file, but the data in the system image file was "back" from its defined position by the difference between the end of the text segment and the next 4K byte boundary.

There is an alternative method for padding the device driver to a fixed size. Our particular assembler had a bug and would not perform the step in the postfix file of:

```
. = start + padsize
```

so a separate utility was written. The utility read the object module that contained the prefix file and the body of the device driver, and it produced an assembly language postfix file that was subsequently assembled and concatenated by the linker. The postfix file generator made enough padding by referring to a global symbol defined in the prefix file which described the fully padded length of the data and text segments.

5.0 CONCLUSION

The technique is straightforward and most experienced UNIX people can evaluate the problematical issues for themselves. We ran into several problems in our implementation and several others are generic to the technique, but the effort was worthwhile because of our initial ignorance of UNIX and its I/O subsystem. The capability, together with dynamic tracing of device driver operation allowed us to research how character and block device drivers operate and how to implement "regular", virtual and asynchronous device drivers.

The technique has less value as the initial experience level of the developers increases; it is easy enough to implement that is can become a standard system offering.

OSx: Towards a Single UNIX System for Superminis

Ross Bott
Pyramid Technology Corporation
1295 Charleston Road, PO Box 7295
Mountain View, CA 94039-7295

OSx: Towards a Single Unix* System for Superminis

Ross Bott
© Pyramid Technology Corporation
Mountain View, CA
ucbvax!shasta!ptuvax!bott

OSx is a dual port of 4.2 BSD and System V onto the Pyramid 90x computer, a high end supermini. OSx is designed to be fully compatible with both 4.2 and System V in a fashion that neither suffers performance penalties from the coexistence of the other. This paper discusses some of the details of this design, both internal to the kernel and at the user interface level, along with some of the problems we faced in its implementation.

The direction of Unix on large machines is at a crossroads: Two major Unix implementations exist, both with distinct advantages and disadvantages: 4.2 offers the virtual memory and disk I/O performance necessary to run Unix effectively on a large machine. System V provides the tools to run Unix within commercial environments, and, with the force of Bell Laboratories and Western Electric behind it, should provide better support and possibly better potential for future development. On the other hand, the large majority of high end superminis continue to run 4.1 or 4.2 BSD because of its performance and features, and because the applications 4.1 users have been building for years were designed for this environment.

It is fairly likely that at some point these two systems will evolve towards a single standard. It is even more likely that this will take several years. During this period of dual standards, we feel that there need not be a barrier between the users of the two systems because of the incompatibilities. The OSx project had several design goals:

(1) To provide a system which could run equally well in either world, allowing installations to take advantage of the unique features provided by applications from both environments, as well as moving transparently between environments.

(2) As networked distributed operating environments become more prevalent, it will be a common occurrence that some machines on a network want to run System V and some 4.2. OSx should provide a gateway by which these machines can coexist and

communicate: Users on the network expecting either environment should get the features and compatibility they expect when communicating to OSx (see Figure 1).

(3) Applications developers can use one environment while developing and testing their application to be fully compatible in the other world.

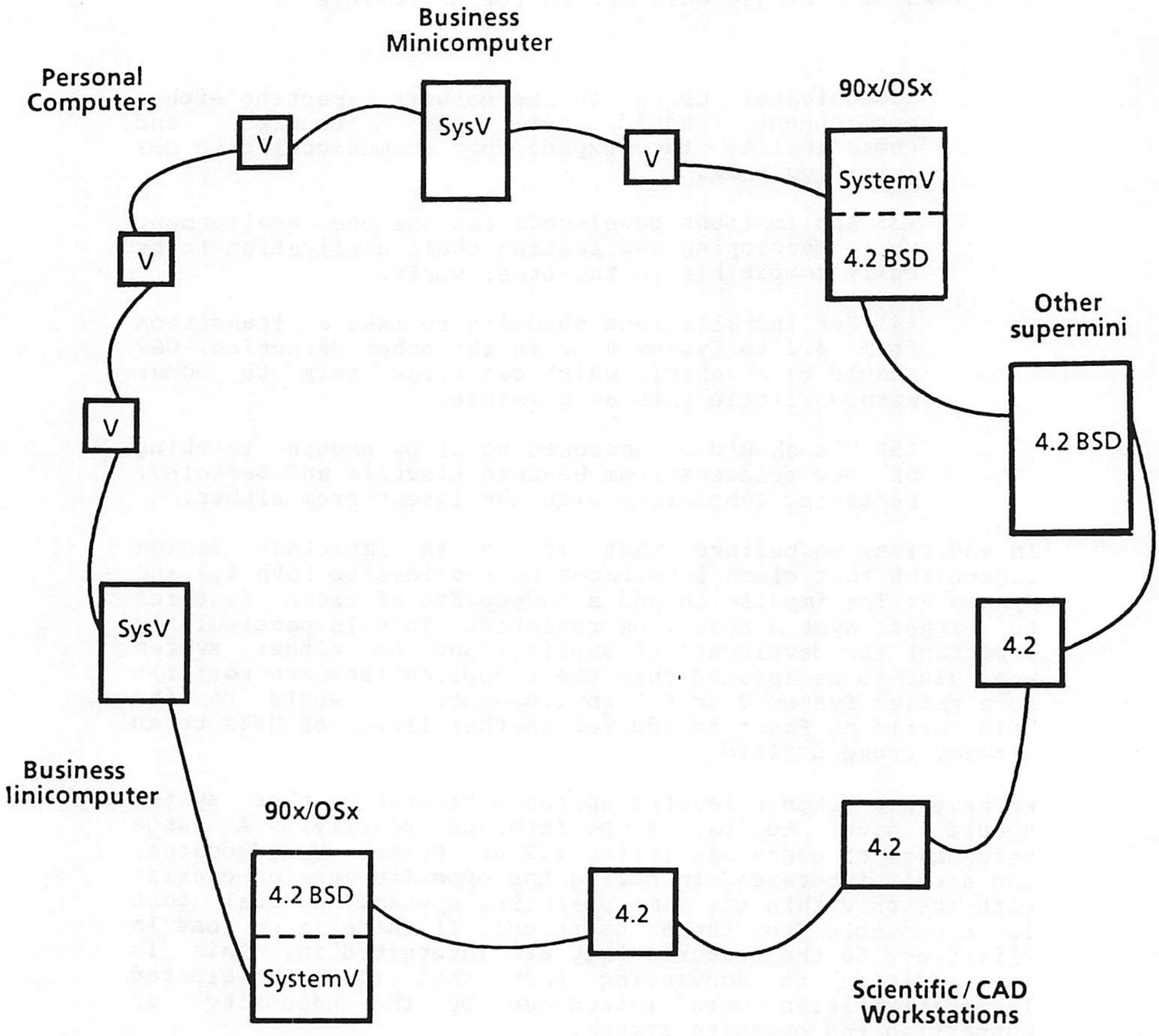
(4) For installations choosing to make a transition from 4.2 to System V or in the other direction, OSx should be a vehicle which can allow this to occur with as little pain as possible.

(5) OSx should be designed to allow smooth tracking of new releases from Western Electric and Berkeley, remaining compatible with the latest from either.

In addition, we believe that it is an important design constraint that clean interfaces be provided to both 4.2 and System V: The impulse to add a hodgepodge of extra features to either system should be resisted. This is particularly important for developers of applications on either system who want to be assured that their applications are portable to a native System V or 4.2 environment. It would do the UNIX world no favor to add yet another flavor of UNIX to an already crowded field.

We have not taken a layered approach because neither system should have to pay a performance penalty. A large percentage of users are strict 4.2 or System V advocates, and aren't interested in having the opposite version coexist with theirs within the same operating system. A dual port is acceptable to these users only if there is no loss in efficiency to the features they are interested in. This is in addition to convincing them that no unanticipated incompatibilities were introduced by the necessity of supporting the opposite system.

Underneath the system interface in a dual port, both systems should be able to take advantage of whichever version provides the best performance and capabilities. For this reason, we are really a 4.2 BSD internally: We believe that 4.2 is clearly superior in performance, for reasons such as the fast file system and demand paging, to be discussed further later. We also feel that some of its features, particularly sockets, provide a useful general basis for other Unix services. (We are aware that Western Electric is "catching up", but high-end superminis can't afford to wait. On the other hand, if future releases of System V (or VI, etc.) contain internal enhancements which provide better



Networked 4.2 and System V Unix Systems

Figure 1

performance, we won't hesitate to incorporate them.)

USER INTERFACE

We had several goals in trying to design an user interface for command execution and program development which would satisfy both 4.2 and System V users:

(1) The directory structure for command binaries, libraries, and header files should look exactly like what users of each system expect. For example, if a command normally exists in /usr/bin, it should still exist there. The decision to keep this standard isn't arbitrary: There are a considerable number of shell files and other applications programs which make explicit or implicit assumptions about the directory locations of programs (as well as a percentage of users with a very firm view of the directory structure).

(2) Commands, libraries, and headers which don't exist in a particular directory for a particular system should not be there. For example, 4.2 users should not see *dircmp* when listing /usr/bin, nor should they see regexp.h when listing /usr/include.

(3) If common commands have conflicting outputs or differing options, users of each type of system should have the outputs and options they expect, and only those. Thus, users of System V should be able to use the -ctime and -cpio options to the *find* command, missing from the Berkeley version. Conversely, 4.2 users should not see these options within their *find*.

(4) All of the above should be accomplished with no significant loss in system performance, and minimized impact on disk usage.

To accomplish the goals above is somewhat analogous to trying to get two different objects to occupy the same location at the same time. What we very specifically did NOT want to do was to "solve" the problem by introducing yet another UNIX version which contained an amalgamation of the features of both of the systems.

The solution we implemented was to introduce the notion of separate 4.2 BSD and System V *universes*, coexisting in the same file structure and sharing a common kernel. At any given time, a user operates in one universe or the other.

The initial universe is determined at login time by an entry in a /etc/u_passwd file. At any point the user can enter the alternate universe by typing one of the commands:

```
warp att (or just att) -- to enter System V
warp ucb (or just ucb) -- to enter 4.2 BSD
```

The commands above actually fork a shell within the alternate universe, and using them is analogous to typing csh while using sh, e.g., one can return to the original universe by typing ^D, or continue to layer alternate universes. To determine in what universe one is operating, one can use:

```
% whereami
You are in the Berkeley 4.2BSD universe
%
```

If a user wishes to execute a single command in the other universe without entering it, this can be done by prefixing the universe, e.g, if one logged in as a System V user, typing:

```
ucb emacs
```

will allow a user to run EMACS within the ucb world. When the user exits, his System V world would be restored. Similarly, from a 4.2 BSD world, typing

```
find . -name "*" -print | att cpio -oaB > /dev/rmt0
```

allows utilization of the System V-only cpio. Note that the inputs and outputs from commands from different universes can usually be piped to each other. Thus, in this case the *find* is a 4.2 version, with the output piped into a System V *cpio*. Wild card expansion and redirection for the whole command is done in the current universe, unless that portion of the command is quoted.

The concept of alternate universes is accomplished by implementing *conditional symbolic links*. These work like normal symbolic links, except that the directory or file to which the symbolic name points is dependent upon which universe the user is in. For example, to set up the /bin directory when the two universes are ucb and att, one would create two separate directories, e.g., /.ucbbin and /.attbin, then use the command:

```
ln -c ucb=/.ucbbin  att=/.attbin  /bin
```

Then, when one is in the 4.2 universe, /bin will look identical to /.ucbbin, both for executing commands and listing the directory. (By using periods before the names, these underlying directories will typically be invisible to users when listing directories.) We also added a new system call, setuniverse, for all users to set and switch the universe in which they are operating. (This system call is used by ucb and att.)

Up to five separate universes can be linked in the fashion above, although we currently plan to use it for 4.2 and System V universes only. These conditional symbolic links work as fast as symbolic links, and have negligible effect on performance. Their effect is further reduced by the command hashing within csh and the soon to be released System V Bourne shell.

We have used this concept for library and header directories as well as the common command directories. A layout of these directories and the directories they are linked to in both universes is shown in Figure 2.

SOFTWARE DEVELOPMENT INTERFACE

The concept of dual universes extends over into the program development area. There is considerable incompatibility in what System V and 4.2 programs expect out of the libraries and kernel interfaces, ranging from the meaning of the return value in printf to radically different mechanisms for handling signals. There are similar conflicts in the /usr/include and other directories of headers. The separate 4.2 BSD and System V directories for libraries and headers allow us to completely maintain these distinctions. The right headers and libraries are automatically accessed by invoking the distinct cc (or f77, etc.) within the desired universe. The children of cc (e.g., cpp, ccom, etc.) will live within the same universe as that cc, and the header and library directories will be as viewed from that universe (see Figure 3).

Once a program has been compiled, however, its behavior with respect to 4.2 or System V compatibility has been permanently established, and the program can be executed in either universe. Thus, the concept of universe must be distinguished from that of application program environment. The latter is a characteristic of each program running on top of the OSx kernel, and can be described by the following aspects:

(1) Program environments are NOT file system or process based. In contrast, a universe is a

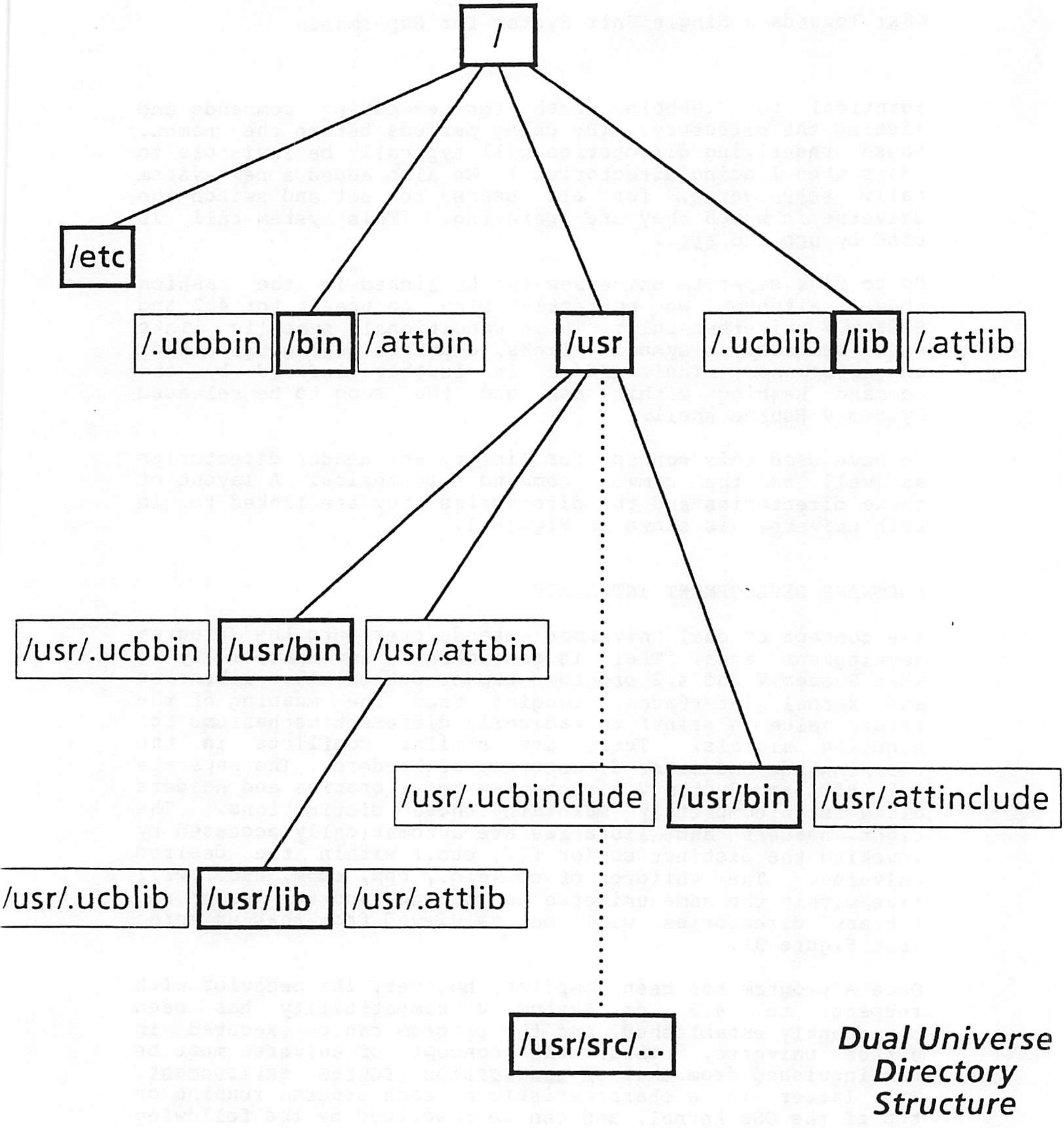
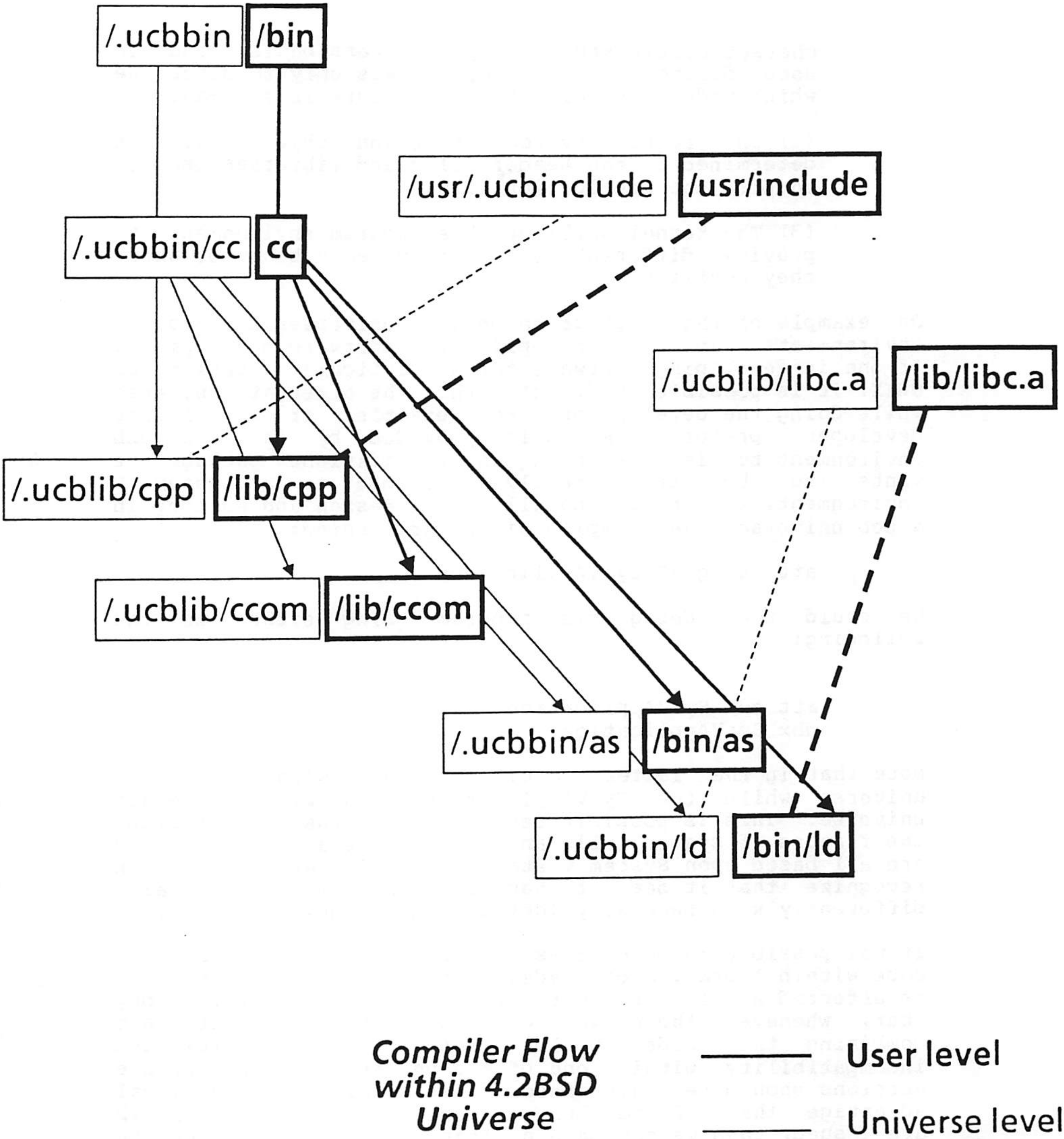

/
/etc
/.ucbbin
/bin
/.attbin
/usr
/.ucblib
/lib
/.attlib
/usr/.ucbbin
/usr/bin
/usr/.attbin
/usr/.ucbinclude
/usr/bin
/usr/.attinclude
/usr/.ucblib
/usr/lib
/usr/.attlib
/usr/src/...
Dual Universe
Directory
Structure

Figure 2



Compiler Flow within 4.2BSD Universe

Figure 3

characteristic kept on a per process basis, and is used during file system access only to determine which underlying directory structure is relevant.

(2) The program environment of an object file is determined by the header files and libraries used to compile the application with.

(3) The kernel utilizes this program environment to provide differential 4.2 or System V behavior where they conflict.

One example of the distinction between universes and program environments can occur in application software development: If one is developing software for a particular version of UNIX, it is possible to "live" within the alternate universe while doing the development. For example, if a software developer prefers the tools provided by the 4.2 BSD environment but is developing an applications package he wants to be transparently portable to a System V environment, then he can do all of his design and editing in a ucb universe, then compile his program using:

```
att cc -g -O SysVApplication
```

He could then debug his program using either of the following:

```
att sdb SysVApplication
dbx SysVApplication
```

Note that in the latter case, dbx runs within the ucb universe while the SysVApplication runs within the att universe. This is possible because, after the compilation, the flavors of system calls and header file assumptions made are all based upon System V standards. The kernel can then recognize that it needs to handle the System V system calls differently when necessary (details in the next section).

It was possible in some cases to modify 4.2 and System V code within libraries or headers such that neither interface is affected at all. However, we've taken the philosophy that, whenever there was even a remote possibility that combining the code would produce an unanticipated incompatibility within one of the universes, then separate versions should be maintained. There is the additional advantage that if and when new releases of System V or 4.2 are issued, then we can make a clean and fast transition to supporting the new versions.

The only exceptions to this philosophy are in areas which do not affect portability and allow jumping between universes without trouble. For example, we support a single object format (BSD style) and have full flexname symbols in either environment. This allows senarios such as the previous example, where debuggers from either universe can be used on a System V object file. (We will, however, offer a flag that can be passed to cc within the System V world which will complain about symbols which are not unique to the first 8 characters, so that applications developers can guarantee portability.)

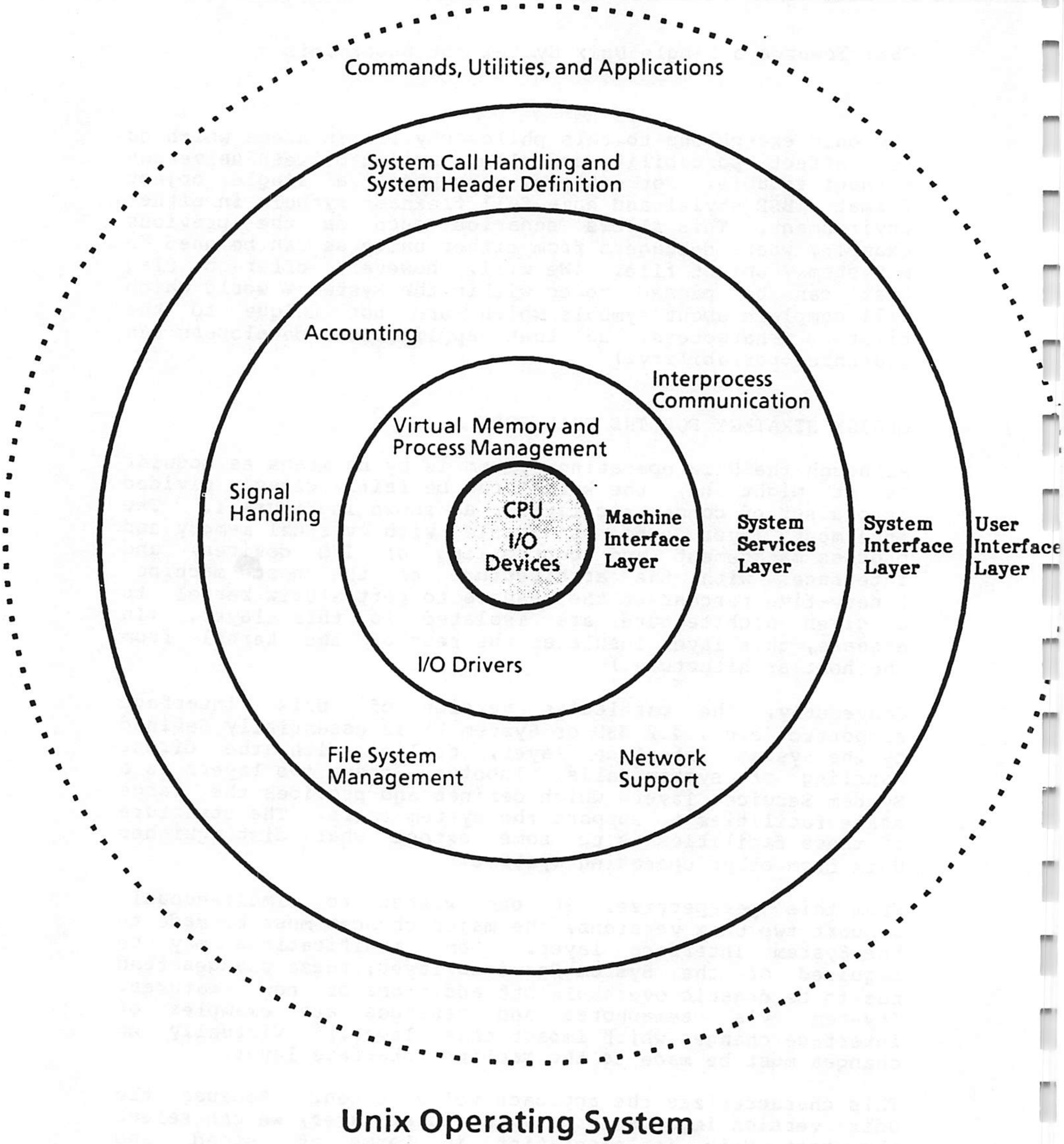
DESIGN STRATEGY FOR THE DUAL PORT

Although the Unix operating system is by no means as modular as it might be, the kernel can be fairly cleanly divided into a set of concentric layers, as shown in Figure 4. The innermost layer concerns itself with virtual memory and process management and controlling of I/O devices, and interfaces with the architecture of the host machine. Ninety-five percent of the changes to port a Unix kernel to a given architecture are isolated to this layer. (In essense, this layer insulates the rest of the kernel from the host architecture.)

Conversely, the particular version of Unix interface supported (e.g., 4.2 BSD or System V) is essentially defined by the System Interface layer, dealing with the direct handling of system calls. Inbetween these two layers is a System Services layer, which defines and provides the large scale facilities to support the system calls. The structure of these facilities is to some extent what distinguishes Unix from other operating systems.

From this perspective, if one wished to simultaneously support two Unix versions, the major changes must be made to the System Interface layer. Some modifications may be required of the System Services layer; these changes tend not to be drastic overhauls but additions of new features. (System V's semaphores and messages are examples of interface changes which impact this layer.) Virtually no changes must be made to the Machine Interface layer.

This characterizes the approach we've taken. Because the Unix version impacts largely the outer layer, we can select the best Unix implementation in terms of speed and capabilities for the internal layers and still support 4.2 BSD and System V. For the inner layers, we've chosen a fairly strict implementation of 4.2 BSD for the following reasons:



Unix Operating System Internal Structure

Figure 4

(1) It is the only Unix version which supports virtual memory and demand paging on large machines. We've modified the 4.2 approach somewhat to take advantage of hardware features available on our architecture, and to provide larger virtual spaces for processes.

(2) It is the only Unix which has really addressed the Unix I/O bottlenecks. For running on a large supermini, we felt the fast file system implementation was a necessity.

(3) Its network facilities are considerably more extensive than other versions. We believe that distributed processing will be predominant in the future, and this network support provides the framework.

(4) There are a variety of other smaller features that are nice to have -- sockets, flexible length file names, more powerful signal mechanisms, etc.

An instantiation of the Unix system structure for OSx is given in Figure 5.

In modifying the System Interface and System Services layers, there are several fairly difficult conflicts to resolve, and a host of smaller changes which are straightforward to implement. Included in the former class are:

(1) Differences in terminal driver design and how character I/O is controlled.

(2) Signal handling

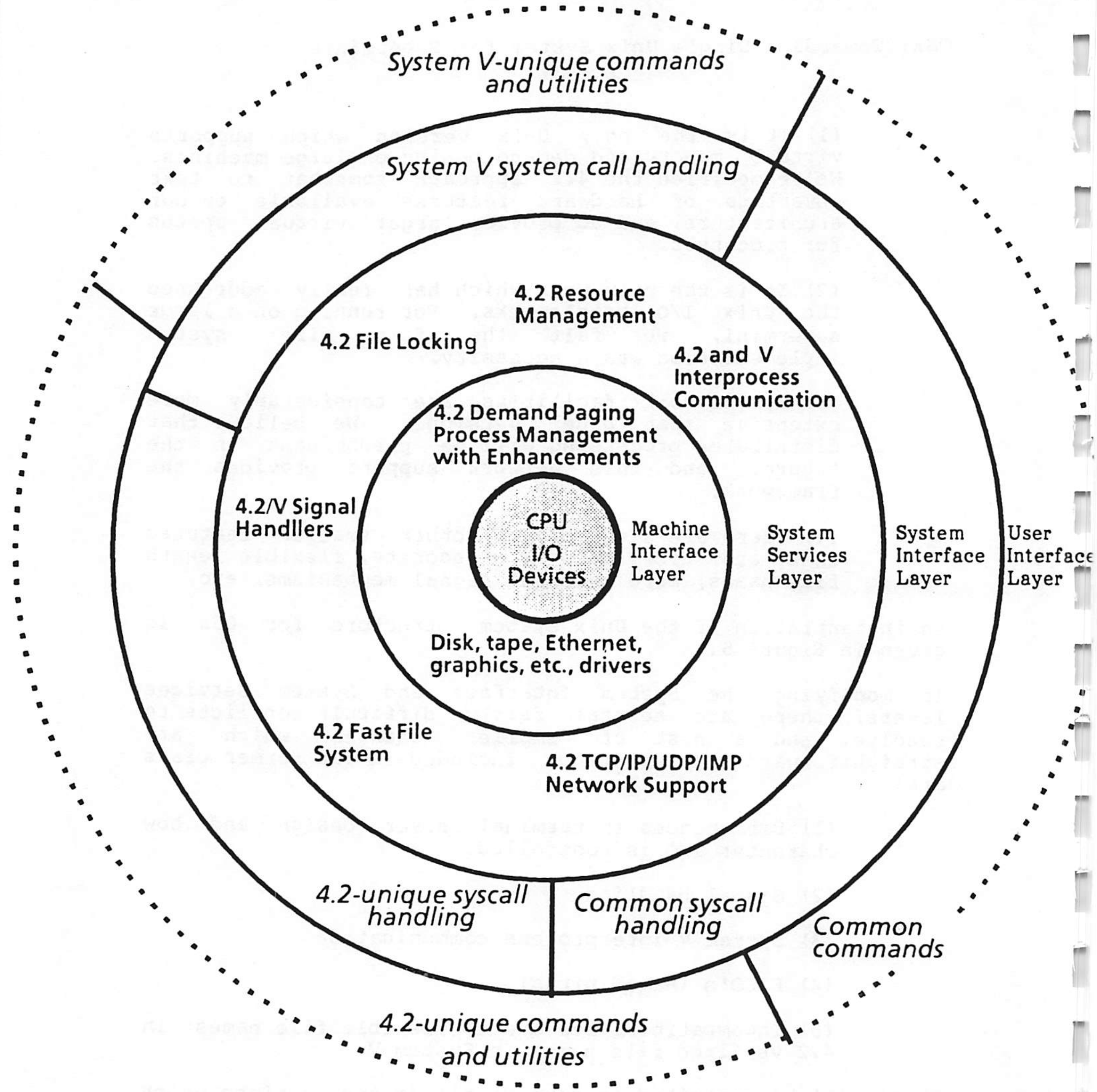
(3) System V interprocess communication

(4) FIFO's (named pipes)

(5) Incompatibilities due to flexible file names in 4.2 vs fixed file names in System V.

These will be described in some detail in the sections which follow. The other changes will be discussed only in terms of the general design heuristics we followed. These included:

(1) System call translation: It is sometimes possible to translate slight conflicts in formats or information returned from a system call from one


System V-unique commands and utilities
System V system call handling
4.2 Resource Management
4.2 File Locking
4.2 and V Interprocess Communication
4.2 Demand Paging Process Management with Enhancements
4.2/V Signal Handllers
CPU
I/O Devices
Machine Interface Layer
System Services Layer
System Interface Layer
User Interface Layer
Disk, tape, Ethernet, graphics, etc., drivers
4.2 Fast File System
4.2 TCP/IP/UDP/IMP Network Support
4.2-unique syscall handling
Common syscall handling
Common commands
4.2-unique commands and utilities

OSx
Internal Structure

Figure 5

Unix version to the other. Because we wanted neither 4.2 nor System V to suffer performance penalties from the coexistence of the other, this "layered" approach is taken only when no significant loss in efficiency was involved. This layering can occur at either the system call stub within libc or within the System Interface layer. Because of the 4.2 basis of our system, all of the translations were from System V to 4.2. Examples of this heuristic are _time_, _ulimit_, _dup_, etc.

(2) Separate 4.2 and System V system call entry points: This is a convenient method of providing to the kernel the knowledge of which version is making the request. This can be passed when necessary to the System Services layer to allow differential handling. Examples of this approach were setpgrp, kill, signal (discussed further below).

(3) Superset structures: System V and 4.2 differ slightly on several system header files (e.g., acct.h, sgtty.h, etc.). In most of these cases, the versions have a common set of structure members with the same names, with each having a few unique members for other functions. By using a system header which is a superset of the common and unique members from each version, utilities and applications programs running under either 4.2 or System V can run transparently to the existence of the other system. Typically, the amount of time required by the kernel to fill the superset structure vs the original structure is insignificant.

(4) Addition of new independent modules: Some of the features of System V are unique enough that it is easiest to incorporate new modules to the System Interface and System Services layers. The major examples of this are the System V semaphores, interprocess message facilities, and shared memory features. It would have been possible to layer these on top of the 4.2 IPC mechanisms, but this was both inefficient and leaves one open to small unanticipated incompatibilities. The amount of additional code to implement these features directly is relatively small, and the impact of interactions on the rest of the kernel is remarkably little.

(5) Special case algorithms: Some conflicts are quite deep. These require designing special mechanisms to handle each in a manner which

maintains compatibility and doesn't sacrifice performance. These are described in the sections below.

To summarize the effect on the kernel of adding System V compatibility, a mirror set of System V system call entry points were added, and the size of the kernel increased by approximately 9%.

DIFFERENCES IN DIRECTORY NAMES (STRUCTURE):

The significant difference here from a systems interface viewpoint is that 4.2 uses a variable length buffer to hold the character names of inodes, while System V expects the directory name buffer to be of fixed length (DIRSIZ).

We believe that 4.2's flexname approach to directories is potentially of great advantage, especially when networked applications become more common, and have kept that full capability in OSx. The problem is then to provide a directory interface to System V utilities and applications compatible with what they expect, while still retaining a flexname directory structure underneath.

The System V directory structure is as follows:

```
struct  direct
{
        ino_t   d_ino;          /* inode number */
        char    d_name[DIRSIZ];
};
```

4.2 BSD uses a variable length structure with an entry, d_namlen, which essentially defines the length of any given directory record:

```
struct  direct
{
        u_long  d_ino;               /* inode number */
        u_short d_reclen;            /* length of
                                      * this record */
        u_short d_namlen;            /* length of this
                                      * d_name string */
        char    d_name[MAXNAMLEN+1]; /* directory string */
};
```

To solve the conflict between the systems, we take advantage of the fact that we know the program environment that the current process is living in. When a process issues a read system call, a flag is set in the process structure indicating that this is a System V read. If the read ends up calling *rwip* (as is the case when reading a directory), this flag is checked and, if this is a System V read, the variable length directory structure read from the disk is converted to a fixed length record which is returned to the user in the following structure:

```
struct  direct
{
        ino_t   d_ino;
        u_short d_reclen;        /* (unused) */
        u_short d_namlen;        /* d_name string length */
        char    d_name[DIRSIZ];
};
```

For applications programs, this is compatible with the original System V structure. We have increased DIRSIZ to be compatible with MAXNAMLEN in 4.2 so that a System V user will never see a 4.2 directory name truncated. This entails no performance penalty in copying the string to user space since the valid characters in a name are terminated by a NULL.

The net effect on System V users is that they can think in fixed length directory records while living on top of a variable length directory file system. (Programs which use the constant 14 rather than DIRSIZ will need to be modified slightly, but these programs should be caught anyway.)

FIFO's (NAMED PIPES)

Named pipes are a feature unique to System V whereby two processes can communicate by a pipe without knowledge of the file descriptor designation at the end of the pipe. 4.2 has implemented pipes in terms of the more general socket IPC mechanism, but did not implement this particular form of pipes.

We've implemented named pipes by adding a new inode type, IFIFO, (a la System V) and using the 4.2 socket mechanism to effect the actual piping, in much the same way that unnamed pipes are currently implemented in 4.2. This entails making changes in the code for opening, closing, reading, and writing from inodes (and utilizing the socket functions *socreate*, *sosend*, *soreceive*, and *soclose*.)

A side effect of this implementation is that 4.2 users can take advantage of named pipes if they so choose. In the interest of a clean 4.2 interface, this feature is unadvertised in the 4.2 world, however.

SIGNAL HANDLING

There are a few major and a variety of minor differences between how 4.2 and System V do signal processing. In terms of signal types 4.2 is nearly a superset of System V, and again provided the starting point for our signal code. The most significant differences are:

(1) In 4.2, signal handlers are set and cleared through a library subroutine, which translates requests into sigvec calls and saves signal masks. In System V, signal handlers are set and cleared directly through a signal system call.

(2) In System V, the function value of SIGCHILD can be set to SIG_IGN. If a wait is then executed, it will block until all children of the calling process have died.

(3) When a signal is caught in 4.2, further signals are held or blocked. Neither option is available in System V.

We've modified 4.2 to provide an additional signal system call. Upon entry into the kernel, a flag is set in the process structure for that process indicating that the process wants System V- styled signal handling. (Note that whether a process invokes System V or 4.2 signal handling is independent of which universe the user is logged into.) For example, wait checks this flag before deciding how to handle SIGCLD.

SYSTEM V INTERPROCESS COMMUNICATION

In the area of IPC, System V introduced three new mechanisms, semaphores, messages, and shared memory, along with the associated system calls. For purposes of efficiency, and to avoid subtle unanticipated incompatibilities, we incorporated these features directly into the kernel, rather than layering them on top of sockets. In the case of semaphores and messages, the code ported almost without change from System V. The shared memory feature required porting to our virtual memory system (we were able to take advantage of hardware support within the 90x for shared memory pages). The first two cases require virtually no changes to the rest of the kernel. Shared memory is a new type of virtual segment, and, as such, must be specially handled by the pager and swapper. However, if such features are not used (as in the case of a strict 4.2 user), there is no impact at all on performance.

TERMINAL DRIVERS AND IOCTL

Providing dual 4.2 and System V compatibility for the operation and control of terminal I/O was perhaps the single most difficult compatibility task in implementing OSx. Although the interface through the *ioctl* system call is essentially identical in the two systems, the degrees of control provided and the underlying implementations are radically different. In this case, neither system is a superset of the other -- each provides a subset of unique features as well as common ones.

Although we make no claims about the ability of a user to freely switch between System V and 4.2 universes without subtle problems arising, we anticipate that users will attempt to take advantage of both universes. Therefore, in addition to supporting both System V and 4.2 views of terminal I/O, we had a secondary goal of allowing users to switch back and forth between universes without leaving their terminal in unexpected states. The potential difficulties here can be illustrated by the following example.

We sign on into a System V universe, and do an *stty* to set the IOCTLs on our terminal to some value. We then decide to enter the 4.2 BSD universe and execute some program there which requires modification of the terminal state (e.g., from cooked to raw mode, as in the case of an EMACS-like editor). This program does a *gtty* to save the state of the terminal, then does a *stty* to set the terminal in the state needed for the program. This brings up the first problem:

> (1) We are now in a 4.2 universe, which does not know about part of the state vector known to the System V universe. Thus, when the *stty* system call above is made, only the 4.2 portion of the state vector will be meaningful.

This can be solved by letting the kernel choose default settings for the rest of the state vector such that the 4.2 terminal will behave in a rational manner. During the execution of the 4.2 program, it may issue additional *sttys* to modify the terminal state. In each case, the non-deterministic part of the state vector can be treated in the same manner, using reasonable default settings. (Note that the kernel must take note of the universe of the process issuing the *stty* to know whether part of the state vector is non-deterministic.)

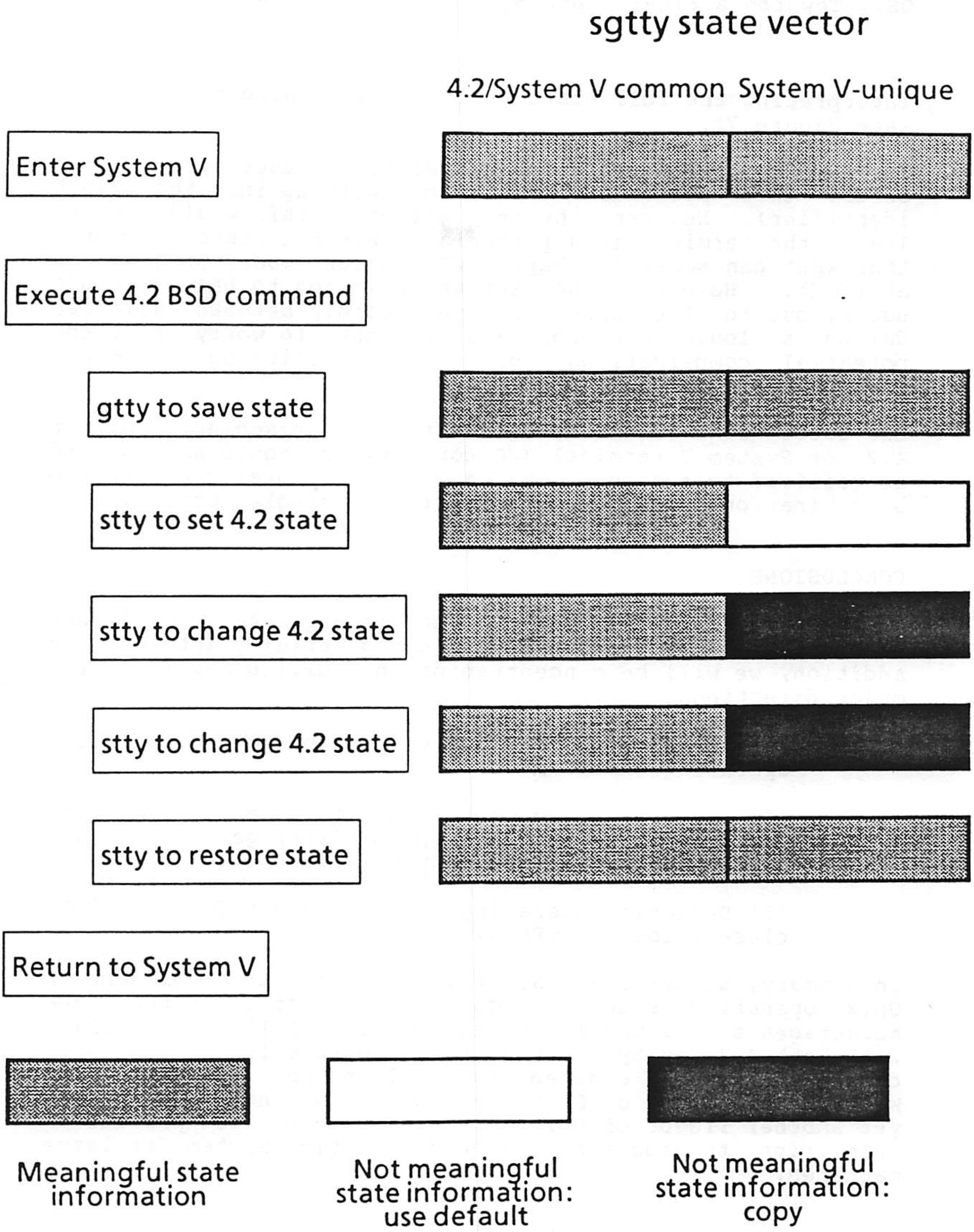
At some point we now exit this 4.2 program and return to our System V environment. Before exiting, the program issues a final stty to restore the terminal to the state it was in when entering the program. This introduces a second, more difficult problem:

> (2) The final stty, by which the 4.2 program intends to restore the System V state of the terminal, differs from all other sttys issued during the life of the program in that the System V portion of the state vector is now meaningful. Yet the kernel has no obvious way of detecting that this state vector is different. If it follows the solution to the first problem, it will choose default settings for the System V-unique portion, destroying that part of the System V environment that the user is returning to (see Figure 6).

Our solution to this particular problem is described as part of what follows.

Although it could certainly be argued that parts of the 4.2 terminal driver are less elegantly designed and written than the System V driver, it has been our experience that the 4.2 version provides significantly better performance, better hooks into hardware flow control, etc. We therefore decided to use the 4.2 driver as a basis. The maze of conflicting and partially agreeing IOCTL parameters were redefined to be non-conflicting (while not changing any of the symbolic names in either universe). This redefinition was done in such a fashion that the terminal driver can detect from the parameter whether the calling process was 4.2 or System V. The terminal state vector within the sgtty structure was modified to include a superset of the states known to the 4.2 and System V universes. Code was added to handle the System V-unique IOCTLs.

Finally, to handle the universe-switching dilemma described above, we've designed a means by which the kernel can detect which stty system calls should be interpreted as transitions between universes as opposed to settings within a single universe: A word is added to the sgtty structure which is transparent to user programs. In it, the kernel puts a 32-bit identifier each time an instance of that structure is passed as part of a gtty system call. This tag identifies the sgtty structure as being from a particular universe. On a stty, the kernel checks this tag. If the identification is not from the universe the process is currently running in, then the kernel interprets the stty as a transition of the terminal state to the opposite universe (with the sgtty structure being passed being the one saved earlier), and interprets the state vector appropriately, introducing partial default settings if returning to 4.2 and



Terminal State Vector During Universe Transitions

Figure 6

interpreting the full state vector if returning to System V (see Figure 7).

It is of course possible for a malicious user program to defeat this mechanism (e.g., by writing into the 32-bit identifier). However, the only effect of this would be to leave the terminal in a potentially strange state (no worse than what can sometimes happen when a raw mode program is aborted). However, the intention is not to bar malicious users, but to allow users to move smoothly between universes during a login session, and not have to worry about the potential complications of the conflicting terminal conventions.

The bottom line is that programs that are based upon either 4.2 or System V terminal I/O conventions should not have to be modified to run under OSx, and that the user can execute a combination of such programs within a single session.

CONCLUSIONS

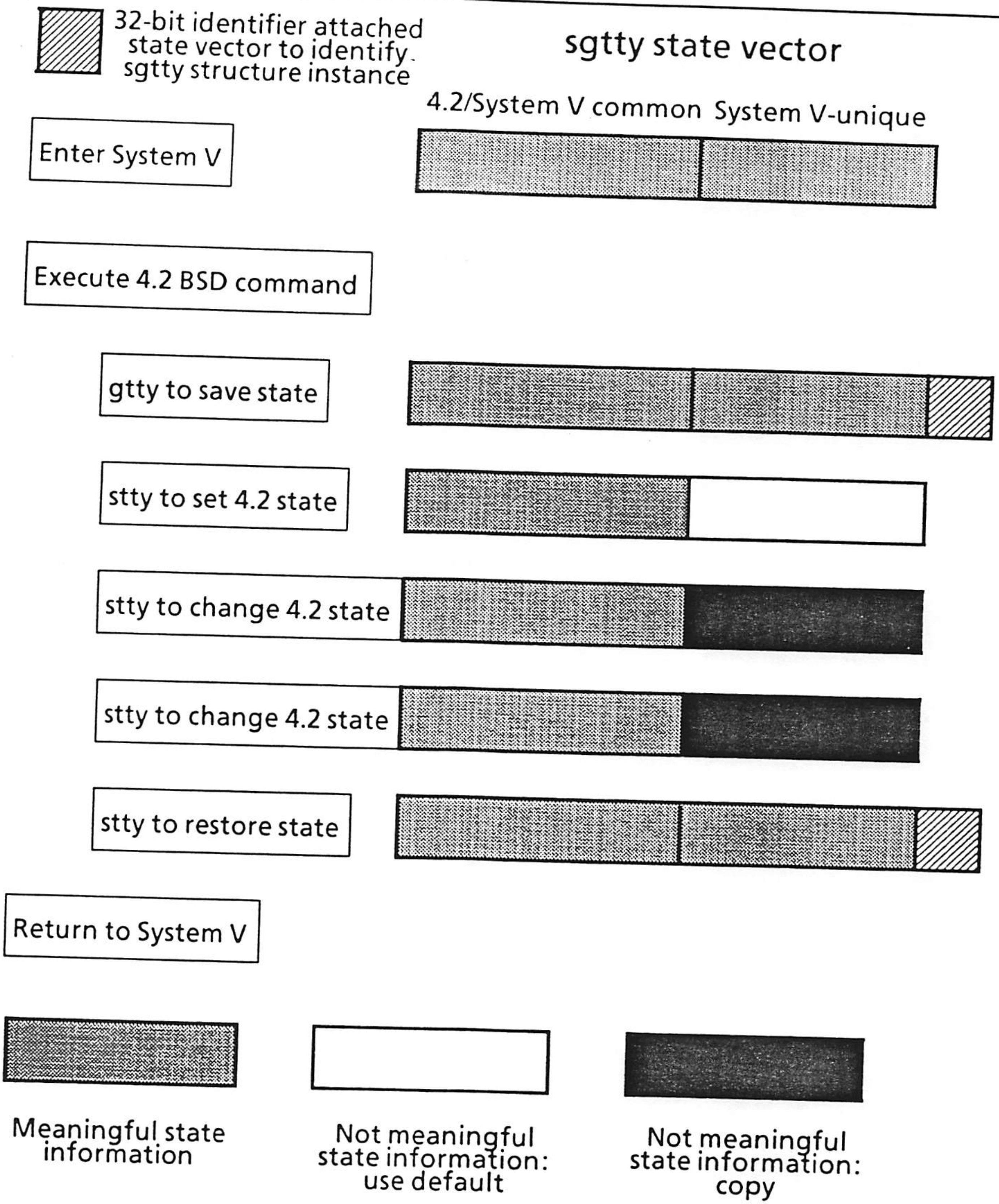
In the future we will continue to incorporate the latest developments from Western Electric and Berkeley into OSx. In addition, we will be concentrating on evolving OSx in three major directions:

(1) Developing a fully networked distributed operating environment.

(2) Incorporating more powerful and efficient virtual memory such as mapped files, general virtual process segments, and copy-on-write forks.

(3) Designing operating system support for multiple closely coupled CPU's.

In summary, we believe that it is possible to implement a Unix operating system which has all of the performance advantages of 4.2 underneath and provides full compatibility with both 4.2 and System V at the interface level, and which can be efficiently updated to reflect new releases from Western Electric or Berkeley. OSx should not be viewed as yet another flavor of Unix but an attempt to provide a transition towards a single Unix operating system for large machines.



▨ 32-bit identifier attached state vector to identify sgtty structure instance

sgtty state vector

4.2/System V common System V-unique

Enter System V

Execute 4.2 BSD command

gtty to save state

stty to set 4.2 state

stty to change 4.2 state

stty to change 4.2 state

stty to restore state

Return to System V

Meaningful state information

Not meaningful state information: use default

Not meaningful state information: copy

Detecting Universe Transitions in *sgtty* Structures

Figure 7

New 1/2-Inch Tape Options and Trade-Offs for 4.25BSD UNIX on DEC VAX Processors

Bob Kridle
mt Xinu, Inc.
739 Alston Way
Berkeley, CA 94710

New 1/2 Inch Tape Options and Trade-Offs for 4.2BSD UNIX* on DEC VAX† Processors

— February 25, 1984 —

Bob Kridle

mt Xinu
739 Allston Way
Berkeley, California 94710

ucbvax!mtxinu!kridle
(415) 644-0146

ABSTRACT

Five tape units covering the spectrum of currently available streaming and start-stop tape equipment were benchmarked under 4.2 BSD UNIX on a DEC VAX 11/750. The performance of each unit was measured in typical UNIX tape applications such as file system backup and tar archiving as well as under optimal circumstances for maximizing data transfer rates. The start-stop systems tested include 45 and 125 ips 1600 bpi units as well as a new, low cost 125 ips, 6250 bpi system. Two streaming tape units were evaluated. One includes a 64 Kbyte cache which allows it to simulate a start-stop unit of similar speed. The other streaming unit, a DEC TU-80, features adaptive mode switching between slow and fast streaming speeds and a slow true start-stop mode.

The performance of all units is reported and compared. It is shown that the new 4.2BSD "fast file system" makes differences in tape unit capabilities more apparent in file archiving applications. A suggestion is made for a modification to the 4BSD tape handler for the TU-80 which could improve its performance significantly.

* UNIX is a trademark of Bell Laboratories.
†VAX, UNIBUS, and MASSBUS are trademarks of Digital Equipment Corporation.

TABLE OF CONTENTS

1. Motivation

2. Equipment

2.1. Cipher Model 100X Drive / Aviv Controller
2.2. Cipher CacheTape Drive / Dilog Controller
2.3. Kennedy 9300 Drive / Wesperco Controller
2.4. DEC TU-80 Tape System
2.5. NEC TD-1505 Drive / DCL Controller

3. Calculated Characteristics

4. Tests and Results

4.1. Maximum Data Transfer Rates Test
4.2. Dump Test
4.3. Tar Test

5. A Proposal for the TU-80

6. Conclusions

Acknowledgements

1. Motivation

The primary purpose of this paper is to explore how users of the DEC VAX series of computers can best exploit the broad range of traditional and new half-inch tape technology now available for these systems. In particular, we are interested in how this technology performs under the newest version of the UNIX Operating System from the University of California at Berkeley, 4.2BSD. Although our tests are confined to this software environment, we feel that many of results may also be useful to those using other software.

Over the last year, important advances have been made in the tape technology available to users of DEC systems. These refinements reflect growth in both basic tape transport technology and in formatter sophistication in adapting that technology to existing applications. These advances can result in significant reductions in cost, improvements in performance, or both, in almost all half-inch tape applications. In some sense, there is a revolution in tape technology taking place now which is similar to that which has already taken place in disk technology, evidenced by the overwhelming acceptance and use of Winchester disk drives.

Important improvements have been made in both streaming and start-stop tape systems. While streaming-tape units with enhancements in both transports and formatters have appeared, start-stop technology has undergone refinement which has produced significant reductions in the costs of both drives and sophisticated 6250 bpi GCR formatters.

The most important changes in the current generation of streaming tape units have been aimed at producing greater compatibility with traditional start-stop tape applications. The first generation of streaming tape units required special software for most applications to produce acceptable efficiency. In addition, if the overall system was not initially designed with the use of one of these units in mind, it was often difficult to integrate a streamer later. If appropriate software was not available, or the overall system could not meet the timing requirements of the streaming transport, degraded performance resulted. Although a "start-stop compatibility mode" was advertised by the vendors of some early streamers, in most cases this was a polite euphemism for "10 to 25 percent performance mode." In this "compatibility" mode, most first generation streamers simply overran the interrecord gap at the end of writing each record and then required a time consuming repositioning operation before the next record could be written.

The two second generation streaming drives we evaluated overcome the time consuming repositioning between each record in different ways. The DEC TU-80 drive uses an unusual transport design which can operate in a true start-stop mode, fully stopping and restarting the tape motion in the space of an inter-record gap. This mode is slower than the best streaming speed, but still acceptable for many applications. When a system cannot generate data at a rate rapid enough for this unit to stream, it simply operates in this start-stop mode. The Cipher M891 CacheTape unit takes an entirely different approach. A 64 Kbyte cache is included as part of the on-board formatter in this drive along with enough intelligence to control the use of this cache in minimizing repositioning operations. We will describe more of the details of these improvements later. Suffice it to say now that they provide considerably better performance when used with systems designed initially for start-stop tape drives than that experienced with the first generation of streaming tape units. They also loosen some of the timing constraints placed on the designs of new systems which will use streaming tape units to reduce cost.

Some significant changes have taken place in the world of start-stop systems as well. In the NEC TD-1505 system we evaluated, these changes have to do mostly with delivering the traditional high performance of a 6250 bpi GCR transport with greater reliability and at a significantly lower cost. Both the GCR formatter and the transport itself have been designed successfully with these goals in mind.

Prices of traditional systems have also come down in response to economies of production scale and heavier competition in both the drive and formatter/interface areas. We have included the Cipher 100X and Kennedy 9300 systems as examples of traditional medium (45 ips) and fast (125 ips) 1600 bpi start-stop tape systems which are of proven reliability and are now available at remarkably low costs.

We have taken each of these five 1/2 inch tape systems and subjected them to a series of tests to determine their effectiveness in a 4.2 BSD UNIX software environment on a DEC VAX 11/750 computer. Most of these tests use the same disk back-up and file archiving software used in production on this type of system. We have included traditional tape systems as well as newer ones so that readers can find points of comparison with familiar existing equipment.

In addition to the test results, we have included some calculated information about capacities per reel at various densities and blocking factors. We feel that this information is also helpful in choosing the most cost effective tape system.

2. Equipment

The tape systems used in this evaluation were chosen to span the largest part of the spectrum of available technology at the time the tests were made. We wished to cover and compare the full range of start-stop technology, from medium speed tension arm 1600 bpi units to the sophisticated, more expensive, high speed 6250 bpi systems. We also wished to examine what seemed to be the most feasible streaming solutions. In addition to being an interesting new technological approach to streaming tape systems, the TU-80 was included because it is offered by DEC directly and is therefore attractive in those applications where it is desirable that all equipment come from one vendor.

Most of the equipment tested here is not unique. In almost all cases competitive equipment with approximately the same characteristics can be provided by several other manufacturers. We intend no particular endorsement of any manufacturer here. Instead we evaluate technology and leave the choice of vendors to the reader. In particular, we can not make any statements about the proven reliability or maintainability of these systems as they are all relatively new to us.

Table 1 Provides a summary of the characteristics of the tested equipment.

2.1. Cipher Model 100X / Aviv TFC-822 Formatter

This tension arm tape transport is one of the most commonly used low cost, medium speed transports. It has been in production for many years and is unlikely to exhibit any heretofor undiscovered bugs or problems. The Aviv UNIBUS interface includes a NRZ (800 bpi) and PE (1600 bpi) formatter. This system had the unique distinction of arriving at our site with a set of keyed and unambiguously labeled cables.

2.2. Kennedy 9300 / Wesperco Formatter

A somewhat more expensive and complex transport than the Cipher 100X, the Kennedy 9300 uses vacuum columns to provide fine tape control at a higher speed (125 ips.) In addition, a built-in exerciser is included to help set up and maintain the tape motion system. The Wesperco UNIBUS interface is similar in function to the Aviv unit mentioned above in that it is a single hex board with included NRZ and PE formatter.

2.3. NEC TD-1505 / DCL Coupler

This tape unit is the top of a line of NEC ½ inch tape transports recently introduced in the United States by DCL, a Japanese controller manufacturer and subsystem integrator. DCL told us they felt that NEC could become the Fujitsu of the tape world in terms of quality and overall value. If our experience is any indication, they may be right. Over the period of our evaluation, we received two of these units, shipped directly from Japan. Both of these auto-loading 6250 bpi GCR tape transports worked immediately after uncrating and installing. No head alignment or adjustment of the auto-loading mechanism was necessary. We did note that in order to operate correctly at the 6250 bpi density, the DCL UNIBUS controller needed

Test Unit Characteristics					
Tape Drive	Cipher 100X	Cipher M891	Kennedy 9300	DEC TU-80	NEC 1505
Interface	Aviv TFC 822	Dilog DU132	Wesperco TC-131	DEC TU-80	DCL
Relative Cost	Lowest	Second	Third	Fourth	Highest
Emulation	TM-11	TS-11	TM-11	TS-11	TM-11
Formatter Loc.	Interface	Drive	Interface	Drive	Drive
Bit Dens.(x100)	8/16	16/32	8/16	16	8/16/62.5
Motion Tech.	Start/Stop	Streaming	Start/Stop	Both	Start/Stop
Tape Control	Tension Arm	Tension Arm	Vacuum	Reel Servos	Vacuum
R/W Speed (ips)	45	50/100	125	25/100	125
Loading	Manual	Auto.	Manual	Manual	Auto.
Rewind speed (feet/sec.)	150	180	>300	190	>300

Table 1.

to located among the first few DMA devices on the bus. Like the Wesperco and Aviv interfaces the DCL controller provides a TM-11 type emulation with which the 4.2BSD UNIX driver can be used. A minor modification is required to support software selection of the three possible tape densities.

2.4. Cipher M891 CacheTape / Dilog DU132 Coupler

The Cipher CacheTape approaches the problem of providing start-stop performance with a less expensive streaming mechanism by including a large RAM buffer in its embedded formatter. The 64 Kbyte cache provides a cushion between the cpu and the tape head. Irregularities in the rate at which data is supplied or consumed by the system are smoothed by this buffer. In addition, data can often be transferred to or from the cache while the tape transport is performing a repositioning operation, effectively masking what would normally be an interruption of the data flow.

A number of thorny problems had to be solved before this relatively simple, elegant idea could be incorporated into a product. For example, how could the formatter avoid finding itself with a full buffer of 64 Kbytes to be written when the EOT tape marker was reached? Cipher has solved this and a number of other problems to produce an effective product.

Another important advantage of this product is its small vertical size. The drive fits in a vertical 8.75 inch space in a standard rack. Tapes are loaded by inserting them horizontally in a slot in the front of the unit. They are automatically threaded toward the takeup spool in the back of the drive. We also observed that this unit autoloads very reliably.

The Dilog interface provided with our evaluation unit emulates the DEC TS-11 interface and is fully supported by the 4.2 BSD TS-11/TU-80 driver. The tape unit supports two speed/density combinations. Either 1600 bpi at 100 ips or 3200 bpi at 50 ips can be selected from the front panel of the drive. The 3200 bpi density is provided to achieve tape economy, but produces little change in performance as the tape speed is halved.

2.5. DEC TU-80 Tape System

The TU-80 is DEC's newest 1/2 inch tape product. Is is the first to be introduced of an anticipated line of tape drives based on the CDC Keystone transport. Like the Cipher M891, the TU-80 uses a horizontally mounted tape transport. However, it is not an auto-loading drive and must therefore be mounted on the top of a short pedestal rack to be accessible for threading.

The TU-80 incorporates several unique characteristics including "adaptive speed/mode selection" and an unusual tape motion control technique. The tape unit formatter shifts the tape transport between three speed/mode combinations automatically to produce the most effective match with the data stream available from the system. These combinations include 100ips streaming mode for the fastest data streams, 25 ips streaming mode for slower data rates, and 25 ips start-stop for the slowest or most irregular data rates between processor and tape subsystem.

The second unusual feature of the TU-80 is its lack of traditional tape motion control mechanisms such as tension arms or vacuum columns in conjunction with capstan pulleys. All tape motion is produced by the transport reel motors. This in turn allows a very short, 13 inch, tape path. The unit is very easy to thread and contains a minimum of moving parts. DEC literature claims the unit requires no regular maintenance other than occasional cleaning by an operator.

3. Calculated Results

The two most important performance measures which distinguish 1/2 tape systems are data density and recording speed. Although we examine both of these measures in this paper, the first, data density, was initially calculated from known constants associated with the particular tape system and then spot checked by testing the number of records of a given size which could be written on a known length of tape. In all tested cases, the calculated densities agreed with those measured to within three percent.

We distinguish here between recording density, a fundamental constant associated with particular technologies and data density, a function of a number of constants including recording density, record size, and interrecord gap size. The equation used to calculate data densities is a follows:

$$average\ density = \frac{blocking\ f\,actor}{\frac{blocking\ f\,actor}{recording\ density} + gap\ length} \qquad (1)$$

We show in Table 2 some of the constants associated with each recording density and the data density achieved with various record sizes. We have also included the percentage loss in inter-record gaps for each recording density and several record sizes.

Notice in particular the gain in capacity of a 2400 foot tape reel recorded at 6250 bpi which can be achieved by increasing the record size. An increase from 10240 bytes per record (which many UNIX programs use by default) to 65536 bytes per record produce an increase in capacity of 24 percent. The added reliability provided by the GCR encoding used at 6250 bpi make records of this increased size feasible.

Also note in Table 1 the 40 percent decrease in reel capacity caused by reducing the record size at 1600 bpi from 10240 bytes per inch to 1024 bytes per inch. As we will see later, a similar reduction in record size is required to optimize the recording speed possible with the

Tape Record Lengths and 2400 ft. Reel Capacities				
burst density (bpi)	800	1600	3200	6250
gap size (in.)	.6	.6	.6	.4
bytes/rec.	record length (inches)			
1024	1.28	.64	.32	.16
10240	12.8	6.4	3.2	1.6
65536	82	41	20	10
bytes/rec.	percent loss in gap			
1024	32	48	65	71
10240	4	9	16	20
65536	1	1	3	4
bytes/rec.	Mbytes/2400 ft. reel			
1024	15.3	23.8	32.0	52.7
10240	21.5	42.1	76.8	147
65536	22.8	45.3	89.8	181

Table 2.

DEC TU-80 tape system when using standard UNIX software.

We also calculated the expected average transfer rates for various recording densities, tape speeds, inter-record gaps, and record sizes. Equation 2 was used to calculate the maximum continuous transfer rates possible for the streaming tape units.

$$\text{average continous data rate} = \frac{\text{speed} * \text{blocking factor}}{\dfrac{\text{blocking factor}}{\text{density}} + \text{gap length}} \qquad (2)$$

Equation 3 was used to estimate the same maximum continuous transfer rate for start-stop transports.

$$\text{average continous data rate} = \frac{\text{speed} * \text{blocking factor}}{\dfrac{\text{blocking factor}}{\text{density}} + \left\{ 2 * \text{gap length} \right\}} \qquad (3)$$

The multiplier of two in the term including the gap size requires some explanation. For start-stop drives, we have assumed that the transport comes to a full stop between each record. Further, we assume that the deceleration and acceleration around stopping between records are constant. This bears close correspondence to the real characteristics of the units for which we have specifications.

If we assume that the tape speed slows linearly to zero and then increases linearly to full speed for next record, in terms of time, this is equivalent to passing through the tape gap at a constant half-normal speed. It is also equivalent to passing through a gap of twice the real length at full speed. We therefore have an effective gap length of twice the real length which can be used in speed calculations.

The results of applying the equations for various densities and speeds are included in the data presented in the next section along with comparable measured data rates.

4. Tests and Results

Tests were performed to measure tape system throughput in a variety of environments. We wished to answer two questions. First, does the tape system perform as predicted by the technical specifications? Second, how does the tape system perform in real applications likely to occur on typical VAX UNIX systems?

In a simplified environment where behavior is expected to reflect only fundamental properties such as tape speed and recording density, we should be able to fully explain our results in terms of these properties. In particular, this type of test was designed to eliminate as many as possible of the irregularities introduced by "real" applications. These tests helped us to gain insight into the basic properties of the technologies.

Basic performance measurements also allowed us to gain confidence in our test setup. They acted as a kind of diagnostic for detecting any installation problems or partial equipment failures which might produce undetected errors in the results of more complex in situ testing. For example, in our first run of basic recording speed tests, one drive performed at a consistently lower level than predicted by its specifications. After further investigation, we discovered that the tape drive used in this test had a malfunctioning playback channel. Although the drive continued to operate using the correction facilities available in the 1600 bpi phase-encoded standard to recover the bad channel, it used many more retries to recover data than would otherwise be required. The lower than expected performance led us to discover and correct the playback channel.

The second phase of testing represents the real motivation for this study. How do these tape systems compare when used in ways representative of the daily operation of most VAX UNIX system. In particular, how do the newer streaming technology and the higher performance 6250 bpi start-stop systems compare against the more traditional units in performing commonplace applications such as file system backup and tape file archiving? The results of both the basic function tests and the applications are reported in the following sections.

4.1. Maximum Data Transfer Rate Test

The maximum data transfer rates possible at a given record size were measured using the simple program below:

```
/* write 500 10k records.c */

char buf[10240];
main()
{
        register i;
        for(i=0;i<100;i++)
                write(1,buf,sizeof buf);
}
```

Standard output was redirected to the tape unit of interest and the time for completion was measured using the built in time function of the command interpreter.

```
% time write_100_10k_records > /dev/rmt0
```

The example above was used for evaluating the maximum possible write data transfer rate to a particular unit. To evaluate the maximum read transfer rate the write system call is replaced by a read call. A constant record size of 10240 was chosen because it is acceptable across all densities and allows direct comparison between them. Results were recorded for several repetitions of the test to assure repeatability. The tape units were also observed to assure correct operation without retries. All tests were run on an otherwise idle system.

The results of the maximum transfer rate tests are reported in Table 3.

Raw Transfer Rates - Estimated and Measured
Measurements Made over 500 10K Records

Tape Drive ips/bpi	Cipher 100X 45/1600	Cipher M891 100/1600	Kennedy 9300 125/1600	DEC TU-80 25/1600	NEC TD-1505 125/6250
Estimates					
Calculated Time(sec.)	83	35	31	138-35	10
Ratios	8.6	3.6	3.1	14.1-3.6	1
Data Rate (Kbyte/sec.)	61	140	160	30 150	530
Read Measurements					
Measured Time(sec.)	84	36	34	62	15
Ratios	5.6	2.4	2.3	4.1	1
Data Rate (Kbyte/sec.)	60	140	150	80	330
Write Measurements					
Measured Time(sec.)	86	35	33	62	13
Ratios	6.6	2.7	2.5	4.1	1
Data Rate (Kbyte/sec.)	60	140	150	80	380

Table 3.

Results are presented in three forms for each drive system and for data rate estimates as well as reading and writing tests. The three forms are: time to complete the test (raw results), equivalent Mbytes per second, and a normalized ratio. The normalized ratio result is calculated for all tests. The time yielded by the test on the NEC 6250 bpi tape drive is always normalized to one and then the same factor applied to the other raw results. For example, the normalized result for the read measurement on the Cipher 100X system can be read as follows: "The Cipher 100X took 5.6 times as long as the NEC TD-1505 to complete the same number of read operations."

Several of the results of these tests require further explanation. In particular the results for the DEC TU-80 system and the NEC 6250 bpi system are not immediately obvious. In both cases the measured values deviate significantly from the estimated value.

In the case of the TU-80, the measured data rate actually falls between the minimum and maximum estimates. The minimum estimate for the TU-80 assumed that the unit would run in start-stop mode at 25 ips throughout the test. The maximum value assumes that the unit will stream at 100 ips for the duration of the test. In fact the unit alternated between streaming at 25 ips and streaming at 100 ips producing data rates between the estimated values.

The NEC 6250 bpi system produced continuous data rates which were consistently less than the estimated values. We have no certain explanation for this. One commonly offered explanation assumes that the location of the controller on the UNIBUS of our VAX 750 restricted the bandwidth available between memory and the tape unit. We rejected this explanation for the following reason. If, at any time during a data transfer operation, data cannot be supplied to or removed from the tape unit, the data transfer must be aborted and retried.

Any restriction of bandwidth between the tape unit and memory will cause "data late" errors and observable tape repositioning operations. We observed no such repositioning. The tape controller contains only 64 bytes of buffering and can not materially effect the available bandwidth.

Our current theory about the discrepancy between observed and calculated data rates holds that we have omitted significant time-consuming operations in our calculation of the maximum possible data rate for a 6250 GCR tape unit. These operations may be associated with the calculation of trailer information such as correction codes at the end of each tape record.

This theory is borne out by two observations. First, the write data rate for the 6250 drive is greater than the read data rate. Second, an indirect calculation of the burst data rate of the drive in the next section yields a result much closer to the calculated value. In both of these cases, we assume that the controller is completing some operation having to do with the previous write transfer after the data transfer is complete. In the case of read operation, the generation and comparison of any error detecting or correcting codes must be done before the read operation can be deemed successful and completed.

In both cases our speculated early return on writes, would allow some overlap between CPU time used by the user program and driver and the completion of the actual write operation. In the case of our raw transfer tests, so little CPU time is required that this overlap cannot be completely utilized; the next tape operation is requested before the previous one is completed and a wait ensues. In the case of the dump test, there is a longer gap between the completion of one write system call and the initiation of another. We speculate that this allows the tape formatter to completely finish any housekeeping generated by one write before the time the next is requested.

4.2. Dump Test

The first of the tests illustrating performance in a typical tape application uses the 4.2 file system backup utility, *dump(1)*. Results are reported in Table 4. below.

Dump Timings 498 10K byte records written					
Tape Drive ips/bpi	Cipher 100X 45/1600	Cipher M891 100/1600	Kennedy 9300 125/1600	DEC TU-80 25/1600	NEC TD-1505 125/6250
Results for Dump Operation					
Time(s)	161	117	82	175(75)	56
Ratio	2.9	2.1	1.5	3.1(1.3)	1
Rate(Kb/s)	31	43	61	28(64)	90
Time(Hours/Eagle)	2.6	1.8	1.3	2.9(1.2)	.93
Results for Tape Portion of Dump Operation					
Time(s)	115	71	36	129(24)	10
Ratio	11.5	7.1	3.6	12.9(2.4)	1
Rate(Kb/s)	44	72	141	38(205)	510

Table 4.

The results in Table 4 were produced as follows: A small file system was created on a a portion of a Fujitsu 2284 160 Mbyte disk drive interfaced to the UNIBUS of our VAX 11/750. This disk drive has transfer characteristics typical of many disks used in VAX UNIX configurations. The file system was then backed-up to tape on each of the tape systems being evaluated. The same 498 10 Kbyte records were written to each tape system. The disk was also dumped to /dev/null (data discarded) to provide an estimate of the time required in the dump operation for work other than tape writes. The dump to /dev/null required 46 seconds.

The first results in Table 4 are the elapsed times required to dump to each tape drive in both raw and normalized form. In the second part of Table 4 the time to dump the file system to /dev/null is subtracted from the results in part one to yield a time required for the tape part of the dump operation. This result is then expressed in normalized form and as an equivalent burst Megabytes per second. This is a measure of the time required per record to write records separated by significant intervals of other activity. Note that this is not a continuous dump rate. It is reasonable to separate tape activity from the other parts of the dump operation because there is very little or no overlap. Operations to and from raw disk and tape are strictly synchronous in this example.

The data in parentheses in the TU-80 column represent results gained when the drive is accessed through the buffered I/O interface so that writes are cached. A constant data stream is made available to the drive in this way and it runs at 100 ips. This also means that a record size of 2048 bytes is produced on the tape, considerably reducing the effective recording density.

Several particularly interesting results were gleaned from this test. First, the DEC TU-80 will not stream at all when with used with the standard UNIX file backup utility. It is clear that either some other backup method must be selected or UNIX must be modified if this drive is to be used with large record sizes.

It is also interesting that although the Cipher M891 streamer performs very similarly to the 125 ips start-stop drive in the continuous transfer tests, its performance degrades somewhat in the dump tests. The explanation for this result is contained in the Cache Streamer's sensitivity to certain combinations of record sizes and inter-record arrival times. Dump provides nearly the worst case combination of parameters causing the noticed degradation in drive performance. It is likely that this degradation could be avoided by making a change in the record size used by dump; perhaps using 5 Kbyte records rather than 10 Kbyte records.

It is also interesting to note that the NEC 6250 tape unit performs better than might be expected from the raw data transfer rate tests. We have already speculated as to why this drive's burst transfer rate is somewhat higher than its continuous rate.

4.3. Tar Test

Table 5 shows the results of timing a small tape archiving operation using the UNIX tar program. Ninety 10 Kbyte records are written to tape in this operation. The results here are very comparable to those yielded in the backup timings.

Tar Timings
Writing 90 10Kbytes Records

Tape Drive	Cipher 100X	Cipher M891	Kennedy 9300	DEC TU-80	NEC TD-1505
ips/bpi	45/1600	100/1600	125/1600	25/1600	125/6250
Time(sec.)	29	19	16	39	12
Ratio	3.2	1.6	1.3	2.4	1
Rate(Kb/sec.)	32	49	58	24	79

Table 5.

5. A Proposal for the TU-80

Our tests show that when the DEC TU-80 is used with typical applications software such as the file archiving or file system backup programs the drive operates most of the time in 25 ips start-stop mode. Benchmarks we have seen of the same tape system used under the

DEC VMS operating system on a VAX 750 show much better performance. We believe this performance level could also be approached under UNIX with a modification to the UNIX driver for the TU-80.

When the TU-80 is written or read through the UNIX buffered I/O interface, its performance is significantly better. For example, the same file system dump which took 175 seconds can be performed in 75 seconds when the tape is written through the buffered interface. The problem with using the current buffered interface in production is that its use forces a physical record size of 2048 bytes. This small record size results in a 22 percent reduction in the total amount of data recorded per tape. For example, a 2400 ft. reel will hold approximately 42 Mbytes at 1600 bpi if a record size of 10240 bytes is used, but only about 33 Mbytes at a record length of 2048 bytes.

The size of the buffer used by VMS when buffered tape I/O is specified is adjustable, as is the number of available buffers. This same feature could be introduced in the VAX version of UNIX. The costs would be twofold.

First, some additional memory would be used consumed by the system at least while tape I/O was taking place. This memory would be lost either to the common pool used most of the time for disk file caching or it could be permanently allocated to the tape handler resulting in a larger kernel and less memory available for user processes. A rough approximation of the memory required can be obtained by examining the size of the cache contained internally in the Cipher CacheTape. This cache is 64 Kbytes. Even if we make the UNIX tape cache twice this size or 128 Kbytes we would be using only about $150-$250 of main memory at current prices.

The second cost of providing and using a main memory cache for the TU-80 is a direct CPU cycle overhead. This is the cost of copying data from one location in memory to another before it is actually written to tape. The data would have to be moved from the user process to the buffer area in the kernel address space. It is precisely this caching action which allows the user process to continue while the tape unit is unable to transfer data due to repositioning operations. A VAX 11/780 can move data from one location in memory to another at a rate of about 12 Mbytes sec using the *movc3* instruction. We assume an 11/750 can achieve rates of at least 5 Mbytes sec. With this figure as the dominating cost, using a buffered data interface would add about 3.5 to 8 seconds of CPU time to the cost of writing a 2400 ft., 42 Mbyte tape. This is probably a small price to pay for the resulting increase in tape speed.

Some complications arise when a tape unit is used through a cached interface. These are primarily associated with the decoupling of errors at the tape unit from the user application. For example, if the tape arrives at EOT while trying to write a record which was previously queued by a user process, it is often unclear how to return the error to that process or to indicate precisely which record caused the error. Previous experience with the use of the current buffered interface has shown this not to be an insurmountable problem. For example, the file system backup program keeps track of how many records have been written on each tape reel in order to avoid running into EOT.

Some aspects of the implementation of this enhanced buffered interface are not clear to us at this time. One choice, for example, would be to extend the current buffered I/O system in 4.2BSD to handle the variable sized buffers necessary for writing varying sized tape records. Another alternative would be to build a separate buffer management system within the tape handler itself, allocating dedicated kernel memory, and and providing a new set of buffer management routines optimized for tape caching. We hope to examine and further discuss these issues in the near future. We are also soliciting input on solutions to this problem from other current or anticipated users of the TU-80 under 4.2BSD UNIX.

6. Conclusions

Each of the non-DEC tape systems we evaluated falls somewhere near the same price/performance line. These systems cover a price range of about four to one and cover a very similar performance range. We also feel that the DEC TU-80 is a good tape system and will represent an excellent value if 4.2BSD UNIX is adapted to provide a better match to its hardware capabilitys. Although we prefer to see caching for streaming tape units done in the drive itself, we have little doubt that modifications to 4.2BSD providing main memory caching will become available.

The choice of an appropriate tape unit for a 4.2BSD VAX configuration depends somewhat of the scale and use of the system. If the system is small or has little data change activity, the traditional medium speed start-stop drive remains a good choice. For very large systems where file system backup can consume large amounts of human, tape, and CPU resources, a 6250 bpi, 125 ips, autoloading tape system like the DCL/NEC we evaluated can be essential.

In the middle performance range, the choice is not as clear. The system integrator can still go with the proven technology of the traditional vacuum column start-stop units. However, it is clear that many of the problems of the early ½ inch streaming tape units have been solved. They are a very attractive and economical choice in their performance range. In the next year we should see these techniques extended further into the high performance range with the availability of 6250 bpi GCR streaming tape units. We are looking forward to evaluating these units.

Acknowledgements

I would especially like to thank all the participating manufacturers and vendors who made this evaluation possible. Each of them arranged to provide me with the necessary equipment without any specific knowledge of what kind of results our tests would produce. Each had enough confidence in his product to feel that the availability of more information could only improve its success. Even in the cases where flaws in the product were uncovered, the vendors only wished to discover a remedy as soon as possible and provided us with every possible assistance in resolving any kind of problem.

Haim Brill of Aviv Corp. arranged for the use of the Cipher 100X and Aviv controller. Kim Kelly of Cipher got the M891 CacheTape to us. Ron Fenn and Paul Anderson, also of Cipher, helped us extensively in understanding and appreciating how the unit worked. Dan Sullivan and John Thomas of Dilog worked with us in evaluating their DU132 controller. They also patiently arranged to have it repaired after we damaged it in an installation error. John Swan and later Malcomb Krongelb from Digital Equipment Corp. were our contacts for the TU-80. Dan McKinster of MCA Associates in San Jose, Shin Nagafuku of Digital Computer Ltd. and Frank Takeuchi of Toyomenka Inc., all worked together to get us our evaluation unit of the NEC TD-1505 based 6250 bpi system.

I would also like to thank my associates at mt Xinu, UniSoft, and UCB who tolerated downtime, made suggestions, and proofread various drafts of this report. In particular, Ed Gould, worked with me in trouble shooting various initial problems in getting all the units working at the same time and modifying 4BSD to support various features of the tape systems. Alan Tobey, Eric Allman, Kirk McKusick, and Carl Smith read and corrected early drafts. I, of course, am responsible for the errors.

A Layered Implementation of the UNIX Kernel on the HP9000 Series 500 Computer

Jeff Lindberg
Hewlett-Packard
Fort Collins Systems Division
3404 E. Harmony Road
Fort Collins, CO 80525

A Layered Implementation of the UNIX* Kernel on the HP9000 Series 500 Computer

by
Jeff Lindberg

Fort Collins Systems Division
Hewlett-Packard
3404 E. Harmony Rd.
Fort Collins, CO 80525
hpfcla!jbl

OVERVIEW OF HP-UX

An implementation of the UNIX operating system kernel, called HP-UX, has been layered on top of an existing operating system kernel for the HP9000 Series 500 computer. The mapping of UNIX functional requirements onto the capabilities of the underlying OS are presented in this article.

The HP-UX operating system is compatible with Bell Laboratories' System III UNIX, and supports most of the standard UNIX commands and libraries. A number of extensions are available, including:

- FORTRAN 77
- HP Pascal
- C
- HP's AGP 3-dimensional and DGL 2-dimensional graphics subroutines
- Ethernet compatible 10 Mbit local area network
- The "vi" visual editor
- Virtual memory
- Shared memory
- HP's IMAGE data base management system
- Support for multiple symmetric CPUs

Another HP9000 family member, the Series 200, was recently introduced. All references in this article to the "HP9000" refer to the Series 500.

SUN OPERATING SYSTEM KERNEL

When the HP9000 project was begun several years ago, the operating system designers took a different approach than that used on HP's previous desktop computers. Even though the first HP9000 system was to be an extension of the BASIC

* UNIX is a trademark of Bell Laboratories

language system of the 9845 desktop computer, an objective of the operating system design was to allow other languages in later versions of the product. The system software was designed in a modular, layered fashion. A central operating system kernel provides a high level interface to the hardware and machine architecture, while other subsystems provide more specific functions layered on top of this kernel. This operating system kernel, called SUN, is described in detail in another article in this issue.

SUN is implemented mainly in Modcal, an enhanced version of Pascal. Modcal supports information hiding via modules, an error recovery mechanism, and systems programming extensions such as absolute addressing. A small part of SUN is implemented in assembly language.

The SUN kernel itself is not directly visible to the user; instead it relies on upper level subsystems such as BASIC or HP-UX to provide a user interface.

Major Components

The major pieces of the SUN OS kernel are the following:

- ⊕ Memory management
- ⊕ Process management
- ⊕ File system
- ⊕ Drivers
- ⊕ I/O Primitives
- ⊕ Real time clock
- ⊕ Interprocess messages

An unusual feature of the file and I/O system is the ability to add new directory format structures, device drivers and interface drivers. These modules can be added without affecting the existing SUN kernel code.

Missing Components

Some key pieces are missing from SUN by design, notably the human interface and program loader. The BASIC system provides its own human interface code which uses the integrated CRT and keyboard of the Series 500 model 20. HP-UX provides a terminal-style human interface to communicate with the user through the integrated CRT/keyboard as well as through normal terminals. HP-UX and BASIC also provide their own unique program loading facilities.

Series 500 HP-UX KERNEL STRATEGY

The basic strategy of the Series 500 HP-UX implementation is to layer the HP-UX kernel definition on top of the SUN operating system kernel. The exact System III UNIX semantics and syntax are kept, but the HP-UX intrinsics are implemented using SUN kernel support instead of porting Bell Labs' kernel implementation to the Series 500.

A layer of code called the "HP-UX layer" resides just above (and in some cases beside) the SUN kernel, as does the BASIC subsystem. The HP-UX layer performs any necessary transformations between UNIX formats and the corresponding SUN formats, e.g. real time clock format. It calls procedures in SUN whenever appropriate, but still has full access to the hardware and architecture when needed. The HP-UX layer maintains a number of higher level data structures which manage HP-UX user processes and user resources.

This layering strategy has a significant impact on the implementation details of the HP-UX layer. For example, Modcal is used instead of C as the implementation language. However, user level code written for System III UNIX will run on HP-UX, unless it depends on certain internal implementation details such as the directory format structure or invisible internal system data structures.

Benefits

The advantages of this approach for HP-UX on the Series 500 come in two main categories: leverage and opportunities for contribution.

A large portion of hardware dependent code was already written for the Series 500 and its peripherals. By using the SUN kernel, the re-implementation of this functionality was avoided in HP-UX. The modules "stolen" included device and interface drivers (especially significant because of the complexity of HP-IB and the new HP CS80 discs), low level memory management, power-up code, process scheduler, architecturally dependent utility routines, and other machine dependent code.

SUN has a number of features which are not present in native UNIX; these features provide opportunities for HP-UX to make a contribution above and beyond other UNIX implementations. These include real-time performance in the area of interrupt response time and process switching, support for multiple CPUs, reliability in the face of system errors, support for variable-size independently managed dynamic memory segments, semaphores, and low level device I/O capability (e.g. GPIO, HP-IB). Also, the IMAGE data base management system was already implemented on top of SUN for the BASIC system. This code has been ported to the HP-UX environment (for release 03.00) to provide this important HP standard data base capability.

Risks

Since the UNIX semantics were being reimplemented in Modcal code on top of SUN, there was a significant risk of incompatibility with System III UNIX. Thus an extensive validation effort was required to ensure compatibility, and to document known incompatibilities due to implementation details. The validation effort and the actual experience in porting UNIX commands and libraries to the HP-UX system proved the excellent UNIX compatibility of the HP-UX kernel implementation.

Another concern was performance of a layered implementation; the risk was that conversion between SUN format and HP-UX format would increase OS overhead. The experience actually observed after the product was completed was that the HP-UX layer itself is responsible for approximately 10 percent of the CPU time used by the kernel; nearly all of that time is spent doing useful work such as loading programs. This means that the SUN functionality is a fairly good match to the HP-UX requirements, since little time is being wasted on conversion between SUN and HP-UX formats.

Also, since SUN was not originally designed with UNIX in mind, the areas which had been tuned for performance were not necessarily those which would make the greatest contribution to performance in a typical UNIX system.

MATCH BETWEEN SUN AND HP-UX

This section describes the areas of SUN that were changed or augmented in order to support the requirements of HP-UX. Only those areas which are important to mapping the UNIX semantics onto the original SUN kernel are described in depth. Little details are given in HP-UX extension areas, and the function of the HP-UX layer itself is generally straightforward so that details are unnecessary.

Some of the additions mentioned are actually maintained separately by the HP-UX layer development engineers, but the code is considered to be at the SUN level. For example, some of the fork code is really at the SUN level because it deals directly with segment tables and other low level data structures.

File System

There was already a good match between SUN and HP-UX in the hierarchical directory structure of the file system. This existing directory format was modified to fit HP-UX semantics rather than implement the native UNIX disc format in Modcal. The fundamental operations such as read, write, open, close, etc. were already supported in a satisfactory manner in SUN; no significant changes were necessary.

But the file system is the area which required the largest changes in SUN. One of the biggest additions was the support of device files, special files which map devices such as printers or terminals into the same name space as regular files. The SUN file system expected device and file accesses to be requested separately. Special checks had to be made for special file types; the new device file code performs operations for device files equivalent to those originally performed for regular files.

Another large change was support for mounting disc volumes onto a currently on-line directory, so that all accessible files and directories are part of a single directory hierarchy. Again, special code was added to check each directory access; if the directory has another volume mounted on it, the access is redirected to the root directory of the mounted volume.

The third area of major change was file access protection semantics. The UNIX read/write/execute and user/group/other mechanisms used to control access to files were not originally in the SUN file system protection scheme. This could have been added, along with the native UNIX disc format structure, to a separate directory format module, since SUN supports multiple directory format structures. However, the characteristics of the existing format were so close to those desired that the SUN format and protection scheme was adapted to the HP-UX requirements.

Changes were made in the SUN file system to support pipes and FIFO files. In the first pass implementation of HP-UX, pipes were implemented in the HP-UX layer. However, they have been moved inside the SUN file system for performance reasons.

A number of minor HP-UX file system operations had to be added to SUN. These include changing the owner of a file, reading or changing file access modes, and duplicating an open file descriptor.

Some operations are performed in the HP-UX layer. These include parsing multi-level path names, managing the user's open files table, and enforcing file size limits on extending files.

I/O

In the area of device I/O, the existing SUN I/O system was a very good match for the needs of UNIX. Virtually no changes were made to the I/O primitives which provide the interface to the backplane and I/O processor, the bus bandwidth management code, the drivers for interface cards or the disc and tape device drivers.

The major changes came in the internal and external terminal support. The external terminal driver was based on the existing serial interface driver, but added UNIX TTY semantics such as type-ahead, line buffering, mapping carriage return/line feed to newline, and sending the interrupt and quit signals.

The integrated keyboard and CRT device control code was based on the work done for the BASIC system's human interface. But the functional operation of the integrated "terminal" had to be completely redone to be compatible with HP terminals.

Memory Management

Because of the simple memory model of HP-UX, the memory allocation intrinsics are easily supported on most operating systems, including the SUN kernel. The major changes in the SUN memory management system were due to the addition of virtual memory capability, which is an extension rather than a semantic requirement of UNIX.

The HP-UX layer has the responsibility of keeping track of the user's memory usage and deallocating this memory when a process or program terminates.

Program Loading

No explicit function for loading and executing programs is present in SUN, but the underlying support needed is there. The file system is used (with minor changes) to find and read the program file, and the memory management system provides the mechanism for allocation of code and data segments. No major changes were required in the SUN kernel to support program loading.

The HP-UX layer manages shared code segments, which allow multiple processes to share a single copy of the code. The HP-UX layer also handles relocation of code and data segments at load time, and meeting the segment attribute requirements requested by the object file format.

Process Management

The HP-UX process management intrinsics are supported fairly well by the SUN kernel, but two areas required a significant effort: fork and signals.

Fork Implementation

The fork system call creates a new process in the exact image of the calling process. It returns to both the parent and child processes just after the fork call, at the point where the function return value distinguishes the child from the parent. Creating an exact copy of a process is not a typical operation supported by normal operating systems, including the SUN kernel.

At the SUN level, code was added to support the "cloning" of a process. This code runs primarily on a separate system process for two reasons: 1) the parent's stack segment must be quiescent at the time of the copy, since it is very difficult to get an accurate picture of a moving object; and 2) the system process has the required addressability to the new child's memory. The parent process cannot gain the required addressability, since its stack resides in its own private address space, and only one private address space can be in effect at one time. The system process stack is in the shared address space, and so is still valid when changing the private address space.

The cloning operation running on the system process calls lower level SUN procedures to allocate memory for the child process and initialize SUN modules for the new process. It is also responsible for duplicating the contents of the parent's segment table in the child's segment table and creating an exact image of all the parent's segments in the child's address space. Special SUN kernel support is necessary to clone the virtual memory segments.

There is also a special version of the process creation code which creates a child process using an existing stack, and causes the child to inherit attributes from the parent.

The HP-UX layer calls this SUN level code to clone a parent process, and then executes other code at the HP-UX level to initialize the new process. This includes allocating an

HP-UX process control block, copying some fields from the parent's process control block, and initializing other unique fields such as process ID and parent process ID. It also increments use counts on shared objects such as shared code segments and open files.

Finally the HP-UX layer returns the appropriate function value to the parent (child's process ID) and to the child (zero).

Signal Implementation

The implementation of signals, primarily the sending and processing of signals, was a significant portion of the HP-UX layer development. SUN had no explicit support for sending asynchronous signals between processes, but did have most of the tools necessary to implement this feature.

One tool is the ability for subsystems to install trap handlers for most classes of traps possible on the Series 500. Signal processing is initiated by triggering an MI (Machine Instruction) trap in the target process, which causes the MI trap handler to be entered on the next machine instruction executed. The MI trap handler is responsible for processing the signal received, and taking the specified action. This can be calling a user-specified signal handler, terminating the process, or just ignoring the signal.

The process scheduler triggers an MI trap in the process about to be dispatched if a signal is pending. The assured periodic interruption of each CPU by the SUN timer interrupt ensures that even a process which is in an infinite loop in user code can receive a signal. The minor changes made to the SUN level code were: a new field in the SUN level task control block used to log the receipt of signals by a process; and the new code in the process scheduler which checks for pending signals.

If the target process is in a known blocked state, the process is forcibly unblocked. Subsequently the MI trap handler processes the signal.

The HP-UX layer code includes the MI trap handler which processes signals, as well as the code which sends signals to one or more processes. It also handles specification of the action to be taken upon receipt of a specific signal, and blocking until a signal is received.

Other Process Management

The process scheduler met the requirements of HP-UX in the original SUN implementation, but has been improved to allow dynamic process priority adjustment to reward interactive processes. SUN supports the creation of special system processes which can provide specific system services. These system processes communicate with user processes and each other via SUN's mailbox-style interprocess messages. Also, a sophisticated set of semaphore operations is provided for synchronization of all processes in the system. This is especially important in a multiple CPU system; merely disabling interrupts does not ensure exclusive access to a shared data structure, since other processes may be running simultaneously on other CPUs. on other CPUs.

The following process management functional areas are implemented in the HP-UX layer:

- ⊕ Higher level support of fork, such as allocation and initialization of a process control block for the new HP-UX process.
- ⊕ Higher level support of signals, including sending and receiving signals, and specifying action to be taken on receipt of a signal.
- ⊕ Management of user, process and group IDs.
- ⊕ Process termination, including deallocation of resources owned by the user process.
- ⊕ Wait for a signal or for termination of a child process.
- ⊕ Management of HP-UX process control blocks.

Other

The functional areas listed below were completely supported by the SUN kernel, except for those changes noted.

- ⊕ Powerup.
- ⊕ Multiple CPU support.
- ⊕ Trap handling.
- ⊕ Real time clock; the HP-UX layer performs the conversion between SUN time format and HP-UX time format.
- ⊕ Alarm clock; the HP-UX layer creates a system process which wakes up each second to see if any alarm signals need to be sent.
- ⊕ CPU times; a minor change was made to the timer interrupt service routine to increment the CPU time used by the current process.

Tools

The existence of system-software development tools for Modcal and the SUN kernel environment was a significant factor in bringing HP-UX up quickly on the Series 500. Tools available included the Modcal compiler, assembler, linker and other utility programs, as well as a powerful symbolic debugger for use in developing system software. These tools were used in HP-UX kernel development without change.

LIKELY PROBLEM AREAS FOR LAYERED UNIX IMPLEMENTATIONS

The areas likely to cause the greatest grief in layering UNIX on top of an existing operating system are listed below: Those marked with a '*' are likely problem areas even for a 'UNIX-like' implementation which provides only a subset of UNIX capabilities.

- * ⊕ Hierarchical directory structure
- ⊕ Mounting disk volumes onto an on-line directory
- ⊕ Multiple links (alternate names) to a file
- ⊕ File access protection semantics
- * ⊕ Interprocess pipes
- * ⊕ Device files in the regular file name space
- ⊕ UNIX terminal semantics
- * ⊕ Process creation via <u>fork</u>
- ⊕ Signals

Most of the problems likely to be encountered in the above areas were described earlier in this article.

CODE SIZE

The Release 2.0 HP-UX kernel layer includes approximately 45 Kbytes of object code, not counting the 9020 integrated terminal emulator code (approximately 25 Kbytes). For comparison, the SUN kernel contains roughly 175 Kbytes of object code, not including any optional device or interface drivers. These numbers do not include HP-UX extensions to standard UNIX such as IMAGE data base management or networking.

Writing Device Drivers for XENIX Systems

Jean McNamara, Paresh Vaish, Richard N. Bryant
Intel Corporation
5200 N.E. Elam Young Parkway
Hillsboro, OR 97123

Writing Device Drivers For XENIX+ Systems

Authors:

Jean McNamara
Paresh Vaish
Richard N. Bryant

Intel Corporation
5200 NE. Elam Young Pkwy.
HF2-1-800
Hillsboro, Oregon 97123

Abstract:

Today most microcomputers and segments of the minicomputer industries are migrating toward non-proprietary operating systems. XENIX+, Microsoft Corporation's direct port of AT&T UNIX+, is becoming a standard operating system for commercial microcomputers. The most important reason for this success is the open system concept that enables customers to run application packages from a variety of vendors, and migrate to new hardware technologies while protecting their software investment. Peripheral and board manufacturers can add new controllers because the XENIX system provides a means to easily add new device drivers to the operating system.

The paper defines the basic types of XENIX device drivers. The kernel facilities used by device drivers are presented. A brief discussion of system configuration is given showing how device drivers are installed.Next the architecture of a block device driver is described. A similar treatment is given for a character device driver. Finally some driver programming practices are presented along with an example device driver.

Typical block devices are disks and tape, or any device holding a file system where the media is divided into "n" equal sized blocks. Character devices are seen as sequential devices with no innate structure of their own except that data is handled as a series of bytes. All device drivers have the same general structure and interface to the kernel. Each driver has two major parts, the task time routines that are executed by the user process running in kernel mode and the interrupt handler that executes asynchronously. The driver's task time routines are entered indirectly through a data structure known as a device switch which serves to partition the driver from the kernel proper.

1. Introduction Today most microcomputers and segments of the minicomputer industries are migrating toward non-proprietary operating systems. Many commercial customers no longer want to be locked into one company's proprietary software products. Intel's systems group has adopted the XENIX+ system as its standard operating system for the commercial marketplace. The XENIX system is Microsoft Corporation's direct port of AT&T UNIX+ with value added enhancements. UNIX systems have become a standard for commercial microcomputers. The most important reason for this success is the XENIX/UNIX open system concept. Customers can purchase application packages from a variety of vendors. They can migrate to new hardware technologies while protecting their software investment. Peripheral and board manufacturers can add new controllers because the XENIX system provides a means to easily add new device drivers to the operating system.

The purpose of this paper is to discuss some of the major issues in writing a device driver for

+ UNIX is a registered trademark of AT&T.
XENIX is a registered trademark of Microsoft Corporation.

XENIX systems. We can not hope to provide a complete tutorial in one article, but we believe we can present enough information to get an experienced software engineer started. Some familarity with a XENIX/UNIX system would be very helpful but is not required. However, some background in operating system internals and systems programming is assumed. And ofcourse, familarity with the "C" language is helpful.

The paper defines the basic types of XENIX system device drivers. The kernel facilities used by device drivers are presented. A brief discussion of system configuration is given showing how device drivers are installed. Next the architecture of a block device driver is discussed. A similar treatment is given for a character device driver. Finally some driver programming practices are presented along with an example device driver.

1.1. Definition of Device Driver XENIX systems know about two kinds of devices, block and character. Typical block devices are disks and tape, or any device holding a file system. The media is divided into "n" equal sized blocks. The block device driver makes read and write requests to the device through its controller, handling all the device dependent aspects. Character devices are seen as sequential devices with no innate structure of their own except that data is handled as a series of bytes. The character device driver maintains state information about each character device, and manages the hardware aspects of each type of device.

All XENIX device drivers have the same general structure and interface to the kernel. Each driver has two major parts, the task time routines that are executed by the user process running in kernel mode and the interrupt handler that executes asynchronously. The driver's task time routines are entered indirectly through a data structure known as a device switch which serves to partition the driver from the kernel proper.

2. Useful Kernel Procedures The wait and signal functions required for process synchronization on events are provided in XENIX systems by the **sleep** and **wakeup** procedure calls. While waiting for an event, a procedure suspends itself using "sleep." When the event occurs the process uses "wakeup" to signal the sleeping processes. Figure 1 provides an example of the use of these calls.

```
 extern int sem;                         extern int sem;
   .                                          .
/* Wait for condition */
while(sem == 0)                            sem = 1; /* Make available */
     sleep(&sem, PRIORITY);                    wakeup(&sem);

sem = 0; /* Make unavailable */
       .                                      .
       .                                      .
       .                                      .
a) process waiting for event               b) process signaling an event
```

Figure 1: Use of the sleep and wakeup calls.

When a function is to be invoked at some future time, in system clock ticks, the driver uses the **timeout** procedure in the kernel. This procedure has the calling syntax **timeout(fun, arg, tim)** where "fun" is the function which will be invoked with argument "arg" after "tim" ticks of the system clock.

physio is used to do raw character I/O by block device drivers. Direct data transfer from a physical address irrespective of the block format of the media. It is passed the address of the driver's strategy routine , the address of the buffer header, the major/minor device number, and a flag indicating whether a read or a write is to be performed. The calling syntax for physio is **physio(&strat, &buf, device, flag).**

Mutual exclusion is provided through interrupt priority masking by the **spl** function calls. There are eight logical levels from all interrupts enabled **spl0** to all disabled **spl7.** Character devices use "spl5" and block devices use "spl6" for their mutual exclusion. Interrupts at the specified level and lower are masked. The previous interrupt mask is returned by the function so interrupts can be reset at the end of the critical section. Interrupts are reset by a "splx" call using the save interrupt mask as the argument allowing nested critical sections. See the example in Figure 2 to see how this is done.

```
int m;
m = spl5();     /* Disable relevant interrupts */
    .
    .           /* Critical section code */
    .
splx(m);        /* Reenable interrupts */
```

Figure 2. Use of the 'spl' calls in a character driver

disksort is used by block device drivers to place I/O requests on the device's request queue. "Disksort" uses a standard algorithm to sort and queue each request. It takes as arguments the buffer header for the driver and the request and sorts the request into the queue pointed to by the buffer header.

The **lowait** procedure is used to wait for an I/O request on a buffer to complete. A pointer to the buffer is passed as an argument to "iowait." Upon completion of a buffer's I/O request, **iodone** should be called with a pointer to the buffer on which the driver has completed I/O.

deverror is called to indicate to the user that an error has occurred. It is passed three arguments: the buffer header for the device, the command issued and the status returned by the hardware. The action of the routine is to print a message to the user's terminal describing the error.

3. Configuration The building of a XENIX kernel is a relatively straight forward process. In each of the system's code directories–"/sys/h";"/sys/sys"; "/sys/mdep"; "/sys/io"; and "/sys/cfg"–are object code libraries or libraries and source and object files, depending on licensing agreements, and makefiles to build the libraries. The kernel load file is built in the "/sys/conf" directory. Here a makefile executes the system configuration facility "config" to generate the "c.c" file and compile it. Then the loader is envoked which links the library files together producing a "xenix" kernel as its output.

Including a new device driver requires adding the driver to the appropriate libraries and modifying the system configuration files "master" and "xenixconf."

The system configuration facility "config" takes as input "master" which provides the hardware dependencies and configuration information for all possible drivers, and "xenixconf" which specifies what drivers to include and modifiable system parameters. The output is the "C" language source file "c.c."

3.1. Master In the section of the file there is a one line description of the hardware configuration for each possible device driver. An entry must be made for each new device drive.

```
*
*    name   viz   msk    typ   hndir   na   bmaj   cmaj   #   na   vec1   vec2   vec3
*     1      2     3      4      5      6     7      8    9   10    11     12     13
     ixxx    2    0137    04    ixxx    0     0      1    8    0   0002     0      0
```

Description of device definition fields:

1: device name ("ixxx" Intel convention).
2: size of interrupt vector in bytes (usually 2).
3: device mask (usually 0137). Bit mask of which functions used in device switch.
4: device type. 10 for block devices, 04 for character devices.
5: prefix ("ixxx" same as name). Prepended to all device switch function names, i.e., ixxxopen; ixxxclose; and ixxxstrat.
6: not used. Set field to 0.
7: major device number for block devices. This number must be unique. For character devices set field to 0.
8: major device number for character devices. For block devices set to major device number for raw character I/O.
9: maximum number of devices supported per controller.
10: not used. Set field to 0.
11-14: A maximum of four interrupt vector addresses. Field 11 contains the interrupt level for the device. Fields 12, 13, and 14 are for devices using more than one level, otherwise set to 0.

3.2. Xenixconf The modification of this file is considerably simpler by adding a line in the driver declaration part of the file.

```
# prefix        flag
 ixxx    1
```

The prefix is the same as the one found in the "master" file. A one flag causes the driver to be included in the kernel being generated, a zero flag omits the driver.

3.3. c.c Among other things, the "c.c" file contains the block and character device switch tables, and the interrupt call table constructed by "config." These tables define the interface between the XENIX kernel and the device driver.

The entry in the interrupt table corresponds to the interrupt level assigned to the hardware, field 11 in the "master" file. The device switch entry for a block or character device corresponds to the device's major number, fields 7 and 8 in the "master" file. Each device is uniquely known by its major/minor device number. The kernel uses the major number to determine which driver to use and the driver uses the minor number to determine the device. Note that "nulldev" will be supplied for routines that do not exist for a given driver. "nodev" is supplied for table entries for which no driver exists. "nulldev" is a null procedure whereas "nodev" marks an error. "Novec" is supplied to all undefined interrupts.

```
/* Block Device Switch Table */
struct bdevsw bdevsw[] =
{
/* 0 */ ixxxopen, ixxxclose, ixxxstrategy, &ixxxtab,
};

/* Character Device Switch Table */
struct cdevsw cdevsw[] =
{
/* 0 */   nodev,    nodev,    nodev,    nodev,    nodev,  nodev, 0,
/* 1 */ ixxxopen, ixxxclose, ixxxread, ixxxwrite, ixxxioctl, nulldev, 0,
};

/* Interrupt Table */
int  (*vecintsw[])() =
{
    clock,
    novec,
    ixxxintr,
        .
};
```

3.4. Driver Files Information that is specific to the device driver but is still configurable by the user is kept in a file. The driver's private data structures and driver configurable parameters are in the configuration file "/sys/cfg/cxxx.c." The header include file for the driver where constants and data structure descriptions are defined are in "/sys/h/ixxx.h." Driver code is kept in "/sys/io/ixxx.c."

Makefiles in "/sys/cfg" and "/sys/io" will compile the driver files and archive them into appropriate libraries. After archiving the new driver in the libraries and modifying the "master" and "xenixconf" files, a new kernel can be generated. With rebooting the system with the new kernel, the new driver is installed.

Each block and character device requires a device special file (device node). The file contains only the device's major/minor device number. The file is the associative link between the devices' name and its number. A device node is created in the device directory "/dev" using the **mknod** utility. For example, **mknod /dev/rdf0 b 0 7** where "/dev/rdf0" is the special file name, "b" the device type (b: block, c: character), "0" the major number, and "7" the minor number of the device.

The major number of a device is the number in the "master" file in field 7 or 8. The minor number is used by the device driver and not by the kernel. Each device node (device special file) will have the same major number but a unique minor number. Remember that for a block device driver both block and character device nodes are required for each device.

To better understand the inner workings of this process, study the c.c file and the files in "/dev."

4. Block Devices A Block device is one divided into 'n' equal sized byte blocks. In XENIX systems the blocks are 1024 bytes long. Data is transferred to and from the storage media are in these 1K byte blocks and the contents are transfered to user space as needed. Examples of block structured devices are winchester disks, floppy disk, and magnetic tapes.

4.1. Block Device Driver Routines Block device drivers must include a total of five routines to perform block I/O including init, open, close, strat, start, and intr. These routines are entries in the "bdevsw" table with the exception of init, which is in the "initsw" table.

The following section describes the function and format of these procedures. They are the interface between the hardware and the xenix kernel and are both required to be present and to follow the XENIX system's naming convention for driver routines. This convention dictates that each routine start with a prefix such "ixxx" where "xxx" is a number representing the device board.

ixxxinit();

"ixxxinit" is called when the system is booted. It is responsible for recognizing the presence of the device and for initializing the device. This routine is a good place for code that need be executed only once.

ixxxopen(dev, flag);

```
int    dev;    /* device number, major/minor */
int    flag;   /* flag set if write access */
```

An open call to the block device driver is issued by the kernel when a user tries to access a file on the device. The driver routine is expected to complete any initialization not taken care of in "init." This may range from a null procedure to a complex algorithm depending on the hardware and the data structures associated with it.

ixxxclose(dev, flag);

```
int    dev;    /* device number, major/minor */
int    flag;   /* flag set if write access */
```

"close" resets any flags and variables "open" set and generally contains code completing any pending operations. Again, "close" is usually a very simple routine or a null procedure.

ixxxstrat(bp);

```
struct bufh  bp;   /* buffer header */
```

Strat is short for strategy, so named because the XENIX kernel calls it when a user actually requests I/O on the block device and lets it determine how to handle the request in the most efficient manner. The "strat" routine is supplied with a parameter which is a buffer header containing all pertinent information on the request including the request type (READ,WRITE,FORMAT), where the data can be found, where to put it, how much data is to be transferred, and what device unit is to be used. In other words, the kernel takes care of a great deal of the work of addressing and buffer allocation. See the more detailed discussion on the buffered I/O system of XENIX below. Strat's responsibility is to validate the request (eg. make sure the device address is within range) and to queue it in the request queue for that device. The request queue may be maintained in some order using the system routine "disksort."

ixxxstart(dd);

```
struct dev;  *dd   /* pointer to device data table */
```

"ixxxstart" is a required internal routine that is called at task time by "strat" and by "intr" at interrupt time. Its function is to start I/O on the device specified by the parameter to it if possible. If there are no pending requests or if the device is busy, "ixxxstart" returns immediately. Starting I/O involves setting up and sending commands to the device.

ixxxintr(level);

```
int level;                /* interrupt level of board */
```

This routine is called by the XENIX kernel upon receipt of an interrupt signal from the device it services. Upon entering "ixxxintr", all interrupts of lower level than the parameter "level" are disabled. The interrupt routine must verify the signal was truly an interrupt from the device and not just a spurious signal then must perform zero or more device dependent services. (Note that checking for a spurious interrupt is a hardware dependent action and may not be possible on some devices.) Some devices require device information be read before another I/O request can be processed. Some devices require a hardware reset signal. Most devices have a status register or block that can be read to obtain current information on the device (ie. current head position on a disk). If another request is waiting on the device request queue, "ixxxintr" should try to start the next request by calling "ixxxstart."

4.2. Raw Character I/O in Block Devices There are circumstances when a "raw" data transfer would be preferable to the buffered data transfer described. For example, when copying an entire disk to another, the overhead of buffering all the data in blocks just to transfer it untouched is unnecessary. In this case the raw character routines of the block device driver would be used. These routines are entries in the "cdevsw" table previously described and include open and close, (same routines as above, listed in both cdevsw and bdevsw),read, write, and ioctl.

lxxxread(dev);
```
int  dev;  /* device number, major/minor */
```

This routine is called by the XENIX kernel to read data from a device serially. The routine consists solely of a call to the system routine "physio."

lxxxwrite(dev);
```
int  dev;  /* device number, major/minor */
```

This routine is called by the XENIX kernel to write data to a device serially. The routine consists solely of a call to the system routine "physio."

lxxxioctl(dev,cmd,cmarg,flag);
```
int  dev;  /* device number, major/minor */
int  cmd;  /* command, user supplied */
caddr_t cmdarg;  /* pointer to structure in user memory */
int  flag;  /* flag word for corresponding file */
```

Sometimes there are special functions one would like to perform on a block device. A classic example is the "format" function for disks. The "ioctl" routine is included in the device driver to perform these special functions. It is responsible for setting up any structures required by the hardware for the function. In the case of the formatting function a format table may be required. To execute the request, ioctl gets a buffer from the kernel and sets up the buffer fields appropriately. It then calls "strat" to queue the request and from this point on the request is handled no differently than any other.

4.3. XENIX+ Buffered I/O System The XENIX kernel has a buffer system to do I/O on block structured devices. This section will provide a very brief description of this system, for while it is an important aspect of block device I/O, it is not necessary to have this information to write a block device driver.

A cache of buffers, each a fixed size (defined in "h/param.h"; 1024 under XENIX) is maintained and each buffer has a buffer header associated with it. The buffer header is set up by the XENIX kernel upon I/O request from the user. It contains all information necessary to complete the request. Each device has what is called a static buffer header associated with it. A blank buffer header acts as the queue head of the request queue for the device. An I/O request is made by the kernel to the driver by passing the request in the form of a pointer to a buffer header to the "strat" driver routine . The buffers containing the data to be transferred is not passed. "strat" links this buffer header into the request queue for that device. Requests are serviced when "strat" calls "start" which pulls the first request off the queue and starts I/O on it. When the device finishes I/O on this request, it generates an interrupt signal and the kernel calls the interrupt routine in the driver. The last action of "intr" is to call "start" if there are more requests queued.

The XENIX kernel increases the efficiency of I/O by waiting to write data out to the device until necessary, and keeps data read from the device in buffers until the buffers are needed for another request. Thus data that might be accessed several times is kept in a cache and device accesses are kept to a minimum.

A buffer header can be in one of 3 states. If it is in the Free state, it is on a free list maintained by the kernel and is available for use. If is is in the Busy state, it is holding request information and is on a request queue or represents a request being serviced. If it is in the Used state, it contains cached data that has been transferred in a previous, serviced request and is available for use by the kernel when the free list is exhausted.

Data Structures

Typical data structures for a block structured device driver include the following:

Device table: structure containing information about the device. Usually includes a state table reflecting state of the device, drive information, etc.

Drive table: Contains information about a particular device including media information.

Parameter block: Contains information required by the device to execute request.

Configuration table: Actual information on device (see section on configuration)

5. Character Devices All character device drivers are divided into two parts. One part are the machine independent functions known as the line discipline routines, the other part are the machine dependent driver functions.

The line discipline routines provide the functionality to enforce all of the XENIX system's character I/O conventions. A default set of line discipline routines come with all XENIX systems. In general, one set is sufficient for all character device drivers and is shared code to all such drivers. The **TTY structure** is the central data structure used by the line discipline routines. Each tty structure contains the queue pointers and current state information for the device. The structure declaration is found in "/sys/h/tty.h." Each device uses three queues: **raw, cooked** (or canonical), and **output.** The "raw" queue holds unprocessed data direct from the device. The "cooked" queue is holds the transformed input data from the raw queue after special character processing. And, the output queue holds data going to the device. Each device can be placed in two modes of operation **raw** and **cooked.** Raw mode is full 8 bit data without parity. The special character functions are not done, and, the reading process is awakened on each character received. In cooked mode all of the special characters are processed. Parity can be done, and, the reading process is awakened only on receipt of a new line, or a special character.

The machine dependent part is written by the device driver writer. But even its structure is in part predefined by the XENIX system. All character I/O device drivers consist of a small well defined set of functions. "Init" is executed only once by the kernel at system boot. The task time entry points to the driver, code executed by the user process running in kernel mode, are "open"; "close"; "read"; "write"; and "ioctl" (I/O control). The hardware entry is through the interrupt routine "intr."

Features supported by all character drivers include full eight bit data or seven bit data with parity, echo control, and some simple line editing. All XENIX character I/O drivers provide immediate character echo of input; command type ahead; driver stop/start of output on command from program or terminal key; and character translation for upper case only terminals.

5.1. Character Device Driver Routines The required driver functions are "init" to initialize the hardware and the driver's internal data structures; "open", "close", "read", "write", and "ioctl" the task time driver entry points; the interrupt routine "intr"; and the start output routine "start."

lxxxinit();

Like all XENIX device drivers, every character I/O driver has an initialization routine that is called once at system initialization. It determines which boards are present, initializes the hardware and driver internal data structures. It is called through the kernel initialization switch "initsw."

The next five functions are the interface to the driver and are executed by the user process running in kernel mode. They are accessed through the driver's character device switch entery "cdevsw" in the kernel.

lxxxopen(dev, flag);

```
int dev;  /* device number, major/minor */
int flag; /* not used */
```

Each character I/O device can be opened independently of any other. The driver uses the minor number in "dev" to determine which device to open. The open initializes the tty structure for the device, the device hardware, and enables device interrupts.

lxxxclose(dev, flag);

```
int dev;    /* device number, major/minor */
int flag    /* not used */
```

Close is the complement of open. It cleans up by flushing the input and output queues, and disables the device interrupt. The later is important to prevent line noise from causing spurious interrupts.

lxxxread(dev);

```
int dev;   /* device number, major/minor */
```

The driver's read function is called through a read system call. It performs the transfer of data from the tty structure's canonical queue to the user process' data space. The reading process is awakened when the request is fulfilled or one of the special characters causing special processing.

lxxxwrite(dev);

```
int dev;   /* device number, major/minor */
```

The write system call transfers data from the user process' data space to the tty structures output queue. It will also start output to the device if it is idle. If the output queue is at it high water mark, the process is put to sleep until the interrupt portion of the driver drains the queue.

lxxxioctl(dev, comd, addr, flag);

```
int dev;      /* device number, major/minor */
int cmnd;       /* command, user supplied */
caddr_t addr; /* pointer to structure in user space */
int flag;      /* not used */
```

The I/O control system call provides the mechanism for the user process to modify flags and parameters in the tty structure and the device's hardware. There is a range of options from baud rate, parity, echo, raw or cooked mode, and vertical and horizontal motion delays to mention a few. The XENIX system users manual provides a complete description of options.

lxxxintr(level);

```
int level; /* board's interrupt level */
```

The interrupt handler is the asynchronous part of the driver. It is called as a procedure by the kernel the interrupt vector. The vector ID is passed to the interrupt routine via the level, thus providing board identification. The interrupt routine handles data input and output for all devices controlled by the driver. It also provides modem control if modems are supported.

Ixxxstart(tp);
```
struct tty *tp;  /* pointer to device's tty structure */
```

There is the start output function. It is unique in that it is called at task time from write, and asynchronously at interrupt time by the interrupt handler. It transfers characters from the output queue to the device, and processes the special delay characters inserted in the output stream.

Ixxxproc(tp, comd);
```
struct tty *tp;  /* pointer to device's tty structure */
int comd;        /* driver output start/stop command */
```

This internal driver routine comes out of modifications made to the machine independent line discipline routines driver access strategy starting in System III. The function generalizes the driver access interface from the line discipline routines. The address of this function is put in the line's tty structure instead of the address of the start routine, as in V7.0 and earlier versions of UNIX.

6. Programming Practices The XENIX kernel guarantees that the following conditions will always be true during driver execution. The kernel guarantees that during a driver's interrupt handler all interrupts at or below the driver's interrupt priority are masked. When a user process is executing the task time portion of the driver, no other process can be scheduled to run until the current process either gives up the CPU by calling "sleep" or some other function that calles "sleep", or, the process exits the kernel in the normal way.

When writing a device driver for XENIX systems a few basic conventions must be observed. The device switch interface can be abreviated but not changed, so the driver must consist of some proper subset of those functions. Block device drivers must have both block and character interfaces. Anytime an "spl" call is used for mutual exclusion a corresponding "slpx" is required to restore the previous interrupt level. The interrupt handler function can never call "sleep" or any function that calls "sleep." And finally, care must be taken as to what code is executed from the clock's "timeout" function; depending on the hardware, race conditions can result.

7. Conclusion In this paper we have attempted to discuss the major issues of writing a device driver for XENIX systems. The major issues being the definition of the two classes of device drivers; the XENIX system runtime support for device drivers; the configuring of a driver into the XENIX kernel; a functional description of the architecture of the two classes of drivers, including an skeleton driver; and a brief discussion of some of the programming practices. It is hoped this discussion has proved to be a helpful guide to writing XENIX device drivers.

8. Sample Device Driver

```
/*********************************************************************/
/*          *** DRIVER CONFIGURATION PSEUDO CODE ***              */

#include "../h/param.h"
#include "../h/ixxx.h"
#include "../h/buf.h"

int Max_num_boards = 1                   /*Max number xxx boards in system*/

struct ixxxpart PP1{  partition table
        fields would contain info on media used in device
    };

struct ixxxcfg ixxxcfg{    configuration table
        fields would contain configuration information such as -
        board number, interrupt level, etc.
  };

 struct ixxxcdct ixxxcdct{    constant device characteristic table
        fields would contain information about the device media
  };


/**********************************************************************/
/*          *** DEVICE DRIVER SOURCE PSEUDO CODE ***                  */

#include "../h/ixxx.h"
#include "../h/buf.h"
#include "../h/param.h"

extern int Max_num_boards;
extern int device_open[];

/*Called by kernel at boot time to initialize device*/
ixxxinit()
{
        for(i=0; i<= Max_num_boards; i++)
                if(board_present(i)){                   /*hardware specific*/
                        call board_init;
                        printf("board %d present0, i);
                }
                else
                        printf("board %d not found0,i);
}

/*Opens a unit by setting flags, initializing variables or structures*/
ixxxopen(dev,flag)
long dev;
int flag;
{
        device_open[minor(dev)] = 1;            /*record device open*/
}

/*Closes a unit by resetting flags, possibly flushing buffers, or queues*/
ixxxclose(dev,flag)
long dev;
int flag;
{
        bpurge(dev);                /*flush buffers; system routine*/
        device_open[minor(dev)] = 0;    /*reset device open flag*/
}

/*Validates request, queues it on device request queue, tries to start I/O*/
ixxxstrat(bp)
struct buf  *bp;        /*see file buf.h*/
{       struct dev  dd;                 /*device characteristic table*/

        verify_request();           /*hardware dependent*/
        disksort(bp);                   /*queue request*/
        if(device_idle())
```

```
                ixxxstart(dd);              /*start I/O*/
}

/*Starts I/O*/
ixxxstart(dd)
{struct dev  dd;                    /*device characteristic table*/

        set_iopb();                         /*OPTIONAL: set up info device needs*/
                                            /* in a parameter block.*/
        out(iopb);                          /* Output in form required by hardware*/
}


/*Processes interrupts due to access completion, seek end, or spurious causes*/
ixxxintr(level)
int level;
{
        struct buf      *bp             /*buffer header*/
        struct buf      *bufh           /*static buffer header*/

        if(spurious_intr())
                return;
        else{
                if(read(status) & ERRORBITS != 0)
                        if(read(status) & SOFTERROR){
                                retry_ct ++;
                                if(retry_ct < MAX_RETRY){
                                        ixxxstart(dev);  /*restart I/O*/
                                        return;
                                }
                                else
                                        deverror(bp, error_code,read(status));
                        }
                        else
                                deverror(bp, HARDERR, read(status));
                else{
                        iodone(bp);
                        if(bufh ->av_forw != NULL)    /*check for more requests*/
                                ixxxstart(dev);
                }
        }
}
```

Transparent Implementation of Shared Libraries

Curtis B. Downing
Bunker Ramo Information Systems
Trumbull Industrial Park
Trumbull, CT 06609

Frank Farance
Systems Theory Design Corporation
555 Main Street, Suite 705
New York, NY 10044

TRANSPARENT IMPLEMENTATION OF SHARED LIBRARIES

Curtis B. Downing

Trumbull Industrial Park
Trumbull, Connecticut 06609
(203) 386-2674
UUCP: [ucbvax!]decvax!bunker!curtis

and

Frank Farance

Systems Theory Design Corporation
555 Main Street, Suite 705
New York, New York 10044
(212) 355-4422
UUCP: [ucbvax!]decvax!std!ff

ABSTRACT

The authors have designed and implemented a shared library facility for Unix systems. This facility was designed to eliminate, in physical memory, duplicate copies of commonly used subroutines. The main feature of this implementation is that the use of shared libraries is transparent to the application programmer.

Hardware has changed since the original Unix design was implemented; small primary memory and fast disk combinations have given way to increased memory and slower disk combinations. The added available memory, however, does not completely compensate for the slower disk. The shared library system, designed and developed to reduce the impact of the hardware limitations of a small desk-top computer (e.g., lack of enough primary memory to compensate for a slow swap device), could be used to reduce the many copies of "libc" library routines which exist in running processes. Routines like "printf" in a shared library could reduce the overall memory consumption for Unix systems. This shared library mechanism was implemented on a Motorola 68000 based Unix V7 system.

The authors detail the transparency mechanism involved and development requirements (hardware, software, kernel). The compilation and loading of programs using shared libraries is addressed as well as the dynamic linking of libraries to each other and to user programs. An example of a shared library development tool is presented.

TRANSPARENT IMPLEMENTATION OF SHARED LIBRARIES

1 A SHARED LIBRARY

A shared library has the following attributes: (1) it consists mostly of executable code, (2) many processes may have simultaneous access to it, (3) only one copy of the shared library resides in physical memory at any instant.

In our implementation, shared libraries take advantage of the Unix System V shared memory system calls. The shared library facility provides two services: (1) tools for the development of the shared library, (2) the run-time support of the library. Note that a shared library is not just shared memory. System V shared memory was designed to hold data without requiring any interpretation of the data. Constant data (e.g., constant text strings) may be stored within a shared libary.

A shared library must allow simultaneous access by multiple processes. There is not, and cannot be, an exclusion mechanism for subroutines in a shared library. Each process has complete access to the shared library at all times. One of the consequences of this is that the subroutines in the shared library must be re-entrant. All data must be kept on the user's stack. The main purpose of shared libraries is to eliminate duplicate copies of functions within physical memory. The shared library achieves this goal by mapping the same copy of the library function into many processes' address space (a result of the shared memory system).

2 MOTIVATION FOR SHARED LIBRARIES

Our original motivation for investigating shared libraries was threefold. First, shared libraries would reduce the amount of physical memory that was used within our Unix system. Second, shared libraries would reduce the amount of swapping in our system (since less physical memory would be used). Third, shared libraries would allow for the construction of very large programs for our application. Shared library code is not considered part of the application process size. Although not a motivation, a requirement of the shared library system is that its use be transparent to both the user and the application programmer.

The reduction of swapping is important because many micro-based Unix systems use Winchester disks for swapping. Since head seek time and rotational latency are large, swapping becomes a very expensive operation. If the amount of physical memory were reduced, the amount of swapping would be reduced.

3 LINKING METHODS

The heart of shared libraries is the linking mechanism. The linking mechanism resolves references between the user private code and the shared library subroutines. There are three basic strategies that we considered: Static Links, Dynamic Links, and Quasi-Dynamic Links. Each of these strategies is discussed below.

3.1 Static Links.

A Static Link is an external reference that is resolved once at link time for an application program. The linker (the ld program) would be supplied with the addresses of the external references that would not have been able to be resolved. Advantages: (1) the subroutine calls to shared library subroutines suffer no performance degradation, (2) minimal runtime support for the shared library is required. Disadvantages: (1) the shared library must be attached to the same logical address each time the library is loaded, (2) if the shared library changes, all programs that use the shared library must be recompiled.

3.2 Dynamic Links.

A Dynamic Link is an external reference that is resolved for each reference during the execution of the application. Either operating system support or hardware support is required. Advantages: (1) with respect to resolving external references, installing a new version of the shared library is not a problem. Disadvantages: (1) operating system modifications will make the operating system non-standard and the application not portable, (2) hardware modifications are expensive and not portable.

3.3 Quasi-Dynamic Links.

The Quasi-Dynamic Link is an external reference that is resolved the first time a reference is made within a process. A lookup routine is required as runtime support. The shared libary subroutine reference is resolved the first time the routine is called. Advantages: (1) new versions of the shared library are not a problem, (2) no extra hardware or operating system support is required. Disadvantages: (1) a slight performance penalty for subroutine calls to the shared library. The implementation described uses Quasi-Dynamic Links.

4 HARDWARE AND SOFTWARE REQUIREMENTS

A minimal amount of hardware and software support is required for an implementation of shared libraries. However, the standard cc, ld, and System V shared memory facilities do not require additions or modifications.

The only hardware requirement is that some form of memory management be supported. Memory mapping and either segmentation or paging are required to support shared libraries. Actually, this hardware support is required to support shared memory. The shared memory system is used as a base for the shared library facility.

There are two main areas of software support that are necessary for shared libraries. The first is the runtime support which contains lookup routines for resolving external references. The second piece of software is the Library Development Tool. The Library Development Tool serves two purposes: (1) to make a "stub" library which will be included with the application program, (2) to make a shared library from an existing collection of subroutines.

5 AN IMPLEMENTATION OF SHARED LIBRARIES

The following section describes a specific implementation of a shared library based on the information presented above.

Ordinarily, as part of their initialization, applications must link themselves to shared library functions. Each application program requests that all the shared libraries it uses be loaded and then linked to the functions which call them. Loading and linking is accomplished by an ordinary shared library described in the section "Library Support Routines".

Shared libraries (SHMLIBs) are linked by a table of pointers called a link table. The link table is located in the application's data segment. The link table information is produced from data in the header of the SHMLIB being loaded. Application programs must allocate space for the link table.

Each global SHMLIB function is assigned an index in the link table. This index is used by the stub to find the appropriate pointer value. Each local SHMLIB function is referenced internally only and does not require a link table entry.

A file, containing a structure for each SHMLIB available for applications, is required. The structure, in turn, contains a function pointer for each global SHMLIB function and serves as a symbol table entry for the SHMLIB.

Another structure; used as a template for the allocation of the application link table, is also included in this file. This structure contains an entry for each SHMLIB symbol table (sym table).

Example:

```
struct libl (sym table)           struct lib2 (sym table)
       {                          {
       int (*check_keyboard)();   int (*read_disk)();
       int (*write_string)();     int (*write_disk)();
       };                         };

       struct libs (template for link table)
       {
       struct libl libl;
       struct lib2 lib2;
       };
```

Libraries are removed only when the use count for a given SHMLIB is zero and shared memory space is needed. Since SHMLIBs contain functions which are used by many programs, once the SHMLIBs are loaded their chances of being used (i.e., linked to a program) are quite high.

Applications can be debugged and tested without using shared libraries. Stub files can be replaced with .o files produced from the library source file. In this case, no change to the application code is required.

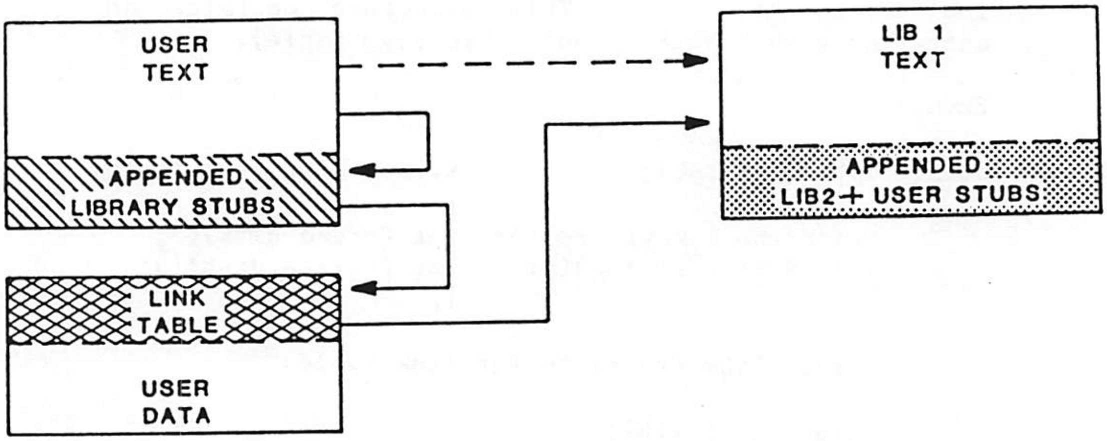
6 STUB MECHANISM FOR LINKING

User references appear to the application programmer to operate as if the functions were located in the user's text segment. The stub has the same name as the referenced external function. SHMLIB references are resolved in ld by a file called a stub file, which is linked with the user files in the ordinary way. The stub objects are in archive form.

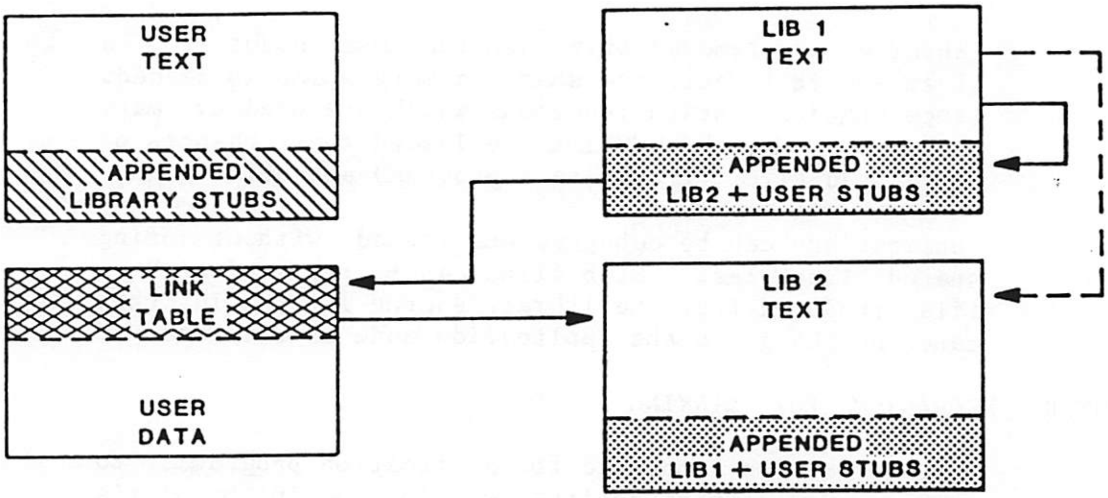
These stubs are analogous to the system call mechanism in the way that it resolves references in the application. The stubs just indicate where the actual function can be found in the same way that system calls tell a trap handling routine which system function to call. The following diagrams illustrate these stub mechanisms.

STUB MECHANISM FOR LINKING

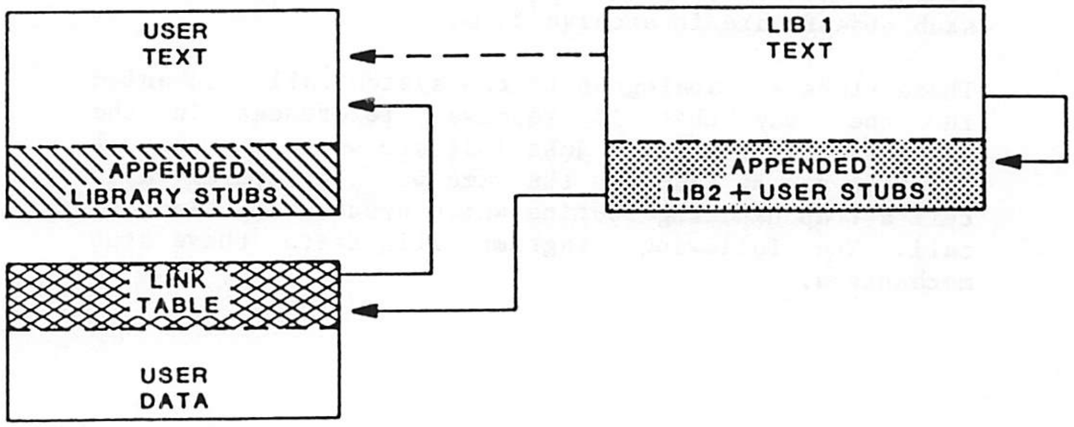
PRIVATE TO LIBRARY CALLS



LIBRARY TO LIBRARY CALLS



LIBRARY TO PRIVATE CALLS



7 LIBRARY DEVELOPMENT TOOL

The Library Development Tool (LDT) is a shell script which produces a loadable SHMLIB file and a linkable archived stub file for each SHMLIB available to the applications.

NAME

tldt -- shell script to prepare libraries for use in shared memory.

SYNOPSIS

tldt file symfile application

DESCRIPTION

Tldt prepares user created shared memory text libraries for loading into shared memory. The file argument is the name of the text library. This library MUST be processed by the script prior to use in shared memory since a shared memory header is inserted at the top of each library.

file

The name of the library to be loaded in shared memory. File is the basename of the text library source. The source for shared memory text libraries is 'file'.c. This name is also the symbol table struct name for that library in symfile.

symfile

A file containing structures (symbol tables) for all shared libraries. Each primary function in a given text library must be an element of a struct in this file.

application

A file which contains the names of each text library needed by that application. Each text library name must be entered in the application file.

The header of a loadable SHMLIB file contains:

- function entry points relative to the beginning of the file and in the same order as symbol table structure entries.
- a count of the number of functions in the SHMLIB.

The LDT uses a sed() script to search a SHMLIB source file for violations of the following shared library restrictions:

- No variables may be declared "static".
- No shared memory system calls; i.e., shmctl(2), shmget(2), shmop(2).
- No sbrk(2) or brk(2) system calls.

8 LIBRARY SUPPORT ROUTINES

The SHMLIB support library, which is also shared, contains: loading, allocating, deallocating, free space management, and removal functions.

The library loading routine requires the name of a file containing a list of libraries to be loaded for this application program. This is the same file which is used by the LDT to resolve undefined references. Library loading occurs with the first load request for the specified SHMLIB. All subsequent requests only update the application's link table and the library use count.

A system table is maintained in shared memory which contains the following entries for each SHMLIB: library name, starting address, and use count.

The library allocation routine allocates on a first-fit basis. If sufficient space is unavailable, libraries with use counts of zero are removed until enough free space is available.

The application's link table is updated by adding the starting address returned from the library allocation to the relative offsets in the library header. The results are placed at predefined positions in the application's link table.

The library deallocating routine decrements the SHMLIB use count but leaves the SHMLIB loaded.

The free space management routine tracks the use of memory in the shared memory segment. The library removal routine is used only by the free space management routine when additional free space is required. Our shared memory segment cannot grow. Its size is determined at boot time and is fixed until the system is rebooted.

We chose to have the shared system library loaded at boot time by a command in the /etc/rc file.

9 LIBRARY PROBLEMS AND LIMITATIONS

The following problems/limitations were found in our implementation of shared libraries:

- External references made in a SHMLIB must use absolute addresses.
- Our version of Unix inserts a function in the application program which is executed prior to the signal handling routine.
- Character strings cannot be specified by " "; you must allocate an array and initialize it with the character string. This is a bug specific to our C compiler.
- Pointers to strings must be initialized with array names only.
- The inaccessiblity of a file pointer when in shared memory prevents the use of fopen(), fclose(), etc. The operating system cannot write the pointer in the appropriate spot. Our solution was to use file descriptors with open(), close(), etc.
- The read system call cannot transfer bytes to an address specified by a character pointer. A character array (buffer) must be allocated in the library and the information then moved from the buffer to the actual library location. Basically, we wrote our own fread. This is a problem specific to our C compiler.

- The open system call cannot contain the file name directly. An array must be allocated and a pointer set to the beginning of the array. The array is initialized with the file name; the pointer is used in the open call. The "file_name" cannot be used in the open call; a pointer must be used in this case. Again, this is a problem specific to our C compiler.

10 CONCLUSIONS

Subroutines belong in a shared library. Many of the subroutines in the "libc" library are duplicated in just about every program. Subroutines like "printf", "scanf" and the standard I/O library are also included in most programs. Using a shared library would save approximately 4K to 8K per process in the system for typical applications.

Another natural for shared libraries is a data base support library, for example. In an application system that involves many application programs, many copies of a data base library could be quite expensive. Basically, any library that is either used often or is very large could usefully be shared.

System calls do not belong in a shared library. The user will not enjoy any savings of space by putting system calls in a shared library. Typically, a system call uses about 10 to 20 bytes of space. The overhead for shared libraries (symbol table space and lookup time) would not save time or space for system calls. Also, routines that are rarely used or are particularly small would not be appropriate for shared libraries.

Performance was considered good. Two extra machine instructions are necessary to execute a SHMLIB function as opposed to a private function. Performance was somewhat hindered because our stub size is approximately 20 to 40 bytes depending on the number of arguments. This is additional text segment code that is not needed for private functions.

Transparency was considered good, although additional code is needed in the application to attach to the shared memory segment and load the libraries. This could have been done automatically the first time the SHMLIB function was called; however, we wanted to avoid mid-process problems and/or failures.

Many of the problems we encountered were caused by deficiencies in our target machine compiler. In most cases, it does not produce relocatable code. Memory mapping hardware limited our implementation of shared memory. Only one pseudo-spare segmentation register was available.

11 ACKNOWLEDGEMENTS

The authors gratefully acknowledge numerous helpful contributions to this paper and the work it describes. Special thanks are due to John DiBiasi, Don Mullis, and Frances Martinez.

UNIX Block I/O Optimization on Microcomputers

David Robboy
Intel Corporation
5200 N.E. Elam Young Parkway
Hillsboro, OR 97123

Unix Block I/O Optimization on Microcomputers

David Robboy
Intel Corporation
5200 N. E. Elam Young Parkway
Hillsboro, OR 97123

January 5, 1984

Introduction

This article considers various optimization strategies for block devices under Unix*, and recommends optimizations which are likely to be most appropriate for commercial microcomputer systems. The objective is to determine what modifications to the Unix kernel and/or file system will produce the best improvement at the least cost.

Concepts

Kinds of Performance Bottlenecks

Data Transfer Rate of Devices In case of large sequential data transfers, such as backups, system performance will be limited by the ability of the device, the controller, and the memory interface to transfer data. In this case we are said to be "I/O bound."

CPU Processing Speed In case of a slow CPU combined with fast devices and complex software, system performance is limited by the processing of operating system requests, and the attendant context switches and process swapping. In this case we are said to be "CPU bound."

Overhead of Access The overhead of each I/O access is a combination of both of the above, plus the time to seek to a track of the disk, and the rotational latency while waiting for a sector. In case of a multi-user system with a great deal of random-access I/O activity, the system throughput can be dominated by seeks and rotational latency, even though, in principle, other processes could be using the CPU during this time. In this case we are said to be "access bound."

Strategies

By way of introduction, several types of optimization strategies are described here. Each of them might be appropriate for a particular kind of I/O bottleneck.

Faster I/O Devices

Faster disks or controllers are a brute force solution in case the system is I/O bound.

Read Ahead

In case the system is access bound, it is desirable to do fewer accesses and throw more burden onto the CPU. Read ahead is useful in case of sequential file I/O, but the worst case involves random access of small records by many users, in which case reading ahead will be useless. However, in practice some accesses are likely to be sequential, such as loading programs and printing spooled files.

* UNIX is trademark of Bell Laboratories

Two read ahead strategies follow.

Larger Block Size Increasing the logical block size means that more data is read or written on each access, thus potentially reducing the number of accesses. However, it also increases the amount of wasted space on the disk. On the average, at least half of a block will be wasted per file. This problem is particularly acute for low end systems with limited disk space.

Track Cacheing Reading a track at a time can save many accesses if more blocks on the same track will be needed, or it can lead to thrashing if many users demand data which is scattered all over the disk. This strategy is likely to be of little use in the current Unix file system, where files are fragmented, but may be of use in conjunction with a strategy that reduces the fragmentation.

Block Cacheing

Block cacheing consists of buffering blocks which have already been read, on the assumption that the same data or other data in the block will be needed again. It is both a form of read-ahead and a method of keeping frequently needed data available. The Unix system already caches blocks, but there is limitless possibility for more sophisticated cacheing algorithms in order to increase the probability that a needed block will be present in memory. There is also the brute-force method of simply storing more blocks.

Organization of the File System

Organizing the disk in order to reduce the number of seeks is a very important strategy. Currently, Unix system files tend to be very fragmented all over the disk, with the result that we are access bound because of seeking. The fragmentation also frustrates attempts at read-ahead because, for example, each block of a file may be on a different track.

This category can include other strategies than just reducing the fragmentation. For example, increasing the block size is a form of read-ahead, but it also has the side effect that a file will contain fewer blocks, therefore the blocks of the file can be found from the inode with less indirection, resulting in fewer accesses to the device.

Seeks may also be reduced by sorting the list of free blocks when files are deleted, so that newly created files will tend to occupy consecutive blocks. Another strategy is to rearrange the order of the partitions on the hard disk, and to rearrange the locations of the inodes within the partitions in order minimize the average distance of seeks.

Direct DMA Transfers

Large sequential reads, such as loading of programs, will be faster if they go directly to the target area of memory, bypassing the buffer system altogether. This also avoids filling all of the buffers with a program, and thereby saves data in the buffers which may be needed again.

Empirical Observations

Current System Characteristics

Wayne Smith of Intel has studied the behavior of Xenix-286** using several histogram tools. We

** Xenix is a trademark of Microsoft Corp. The current version is an implementation of Unix Version 7. Xenix-286 is Xenix running on an Intel iAPX 286 microprocessor.

will very briefly and qualitatively summarize his results here. Using a faster disk controller increased the raw disk throughput greatly, but increased the Xenix system throughput very little for a disk-intensive multi-user application, showing that we are not predominantly I/O bound. Using a faster CPU (the 286 versus the 8086) did not increase system throughput in proportion to the speed of the CPU, showing that we are not predominantly CPU bound. Therefore we are access bound.

This conclusion is supported by two other facts. One is that the kernel histogram shows that we are spending a great deal of time idle; i.e., with all processes waiting for something. The other is that the disk accesses very few blocks between seeks, and it seeks frequently and over great distances.

These conclusions apply to a particular context, in which four to eight processes are doing I/O intensive, random-access disk processing. This is assumed to approximate the situation in a commercial environment, which is our target for optimization. In case of large raw data transfers by a single process, such as a disk backup, almost any system will be I/O bound. On the other hand, we have also seen that for small records (32 bytes), both the system throughput and the amount of physical I/O drop off greatly, and the system is CPU bound.

Berkeley File System Results

Other empirical results that we have available come from the Berkeley Unix "Fast File System." Berkeley claims to have gotten about a tenfold throughput improvement by using essentially two types of strategies. One was a larger block size, and the other was a set of algorithms designed to keep directories and all their associated files and inodes clumped together in local areas of the disk called cylinder groups, in order to minimize seeking. Berkeley did not fundamentally change the Unix file system; their changes can be thought of as optimizations of the implementation, and the performance improvements (if true) are impressive.

The larger block size is useful because, as we have mentioned, it is both a form of read-ahead, and a way of reducing the amount of indirection needed to access a file. Also, it seems intuitively that it may help to reduce file fragmentation by breaking each file into fewer, larger pieces. Its disadvantage is that it wastes disk space, which is a worse problem for microcomputer systems than it was for a VAX* system at Berkeley. This problem was dealt with by allowing a block to contain addressable fragments, so that several small files can occupy parts of a block. This in turn necessitates some dynamic reorganization in case a file outgrows its block fragment.

Keeping the files clumped together in "cylinder groups" was a way of dealing with file fragmentation. Because blocks are allocated according to their location instead of in the order of the free list, this method necessitates maintaining a bit map of available blocks, and some more complexity in maintaining the free list itself. Another complexity is that large files would fill up a cylinder group and leave no room for other files, which would defeat the attempt to clump them together. Therefore as files grow large, additional blocks must be allocated for them outside of their cylinder group.

Optimization Strategies Considered

Speed Up Key Functions in Software

Since we are not CPU bound, there is limited value in improving the CPU performance, but there may be cases where this is justified because it is easy, especially for such things as using machine DMA instructions to move data in memory or clear buffers.

More Buffers in the Block Cache -

The brute-force solution to block cacheing is to have more buffers. As processors and memory become faster and cheaper, the block I/O becomes relatively more of a bottleneck on microprocessor systems, but at the same time brute force solutions such as larger buffer caches become relatively more feasible.

Reducing Access Latency The latency of each access depends on the time to seek to the track, and then the time to wait for the sector to rotate around to the heads. An important method of reducing the number of seeks is to reduce the fragmentation of files, which is covered below. A method of reducing the average seek distance is to rearrange the order of partitions (that is, file systems) on the hard disk and the locations of inodes within the partitions, so as to reduce the average distance of seeks. This is possible independently of the actual implementation of the Unix system. Finally, the rotational latency can be reduced by having the appropriate sector interleave for the particular combination of disk, CPU, and operating system.

Reducing Fragmentation of Files

Sorting the Free List A way to reduce file fragmentation, and thus minimize seeking, is to sort the list of free blocks when deleting files, so that new files will tend to be less fragmented.

Sorting the free list does not guarantee that files will be contiguous, since different tasks may be requesting blocks concurrently, but it should help to reduce file fragmentation.

Cylinder Groups

By the title "cylinder groups" we mean the method of organization used by the Berkely fast file system. As mentioned above, this is rather complex to implement. It is also of questionable value, since in a multi-user commercial environment it is not clearly useful to have the various files in a directory clustered together. Also, although the blocks of a file are kept close together, they are not necessarily kept sequential, so that it would not be possible to do large DMA transfers of files directly into user areas.

On the plus side, this method would keep inodes relatively close to the corresponding files, which would reduce the distance of many seeks.

Periodic Reorganization of the File System

Another way to reduce or eliminate fragmentation is to have a utility that periodically consolidates all files into contiguous blocks (that is, logically contiguous, with an interleave in the disk format). This solution is esthetically unappealing, but it has some advantages. It requires no changes to the kernel, being a utility. It introduces no incompatibilities in the file system. It guarantees that the blocks of files are contiguous, rather than just close together, so that a DMA program loader could load programs very rapidly.

Reorganizing the files would mesh very will with sorting the free list, mentioned above, since it would be easier to find sequences of consecutive blocks if the free list was already sorted. A single utility could do both.

On the minus side, this utility would have to be run periodically by a system administrator in single-user mode. This chore might be neglected unless we run it automatically at bootstrap time, which would make booting the system time consuming. Also, in case the disk is very full, it might be impossible for the utility to do any reorganizing for lack of space. The first objection might be overcome by making the utility a perpetual background task, but this would be tricky with multi-users running; it would require mutual exclusion while the file system is being manipulated. The second objection might be overcome by using the swap space.

The worst case for this solution is probably a situation where there are a few large, permanent files that grow slowly. In this case, reorganizing might be necessary frequently, slow, and impossible on a nearly full disk. A way around this might be to reorganize very large files into smaller clumps of contiguous blocks. Another type of worst case is a situation where many files are often created and erased, such as a software development environment. In this case, the file system would fragment itself rapidly, so reorganizing would be necessarily frequently, and it would have a lot of work to do.

Reorganizing Files When They Are Closed

A variation of the above strategy is to reorganize each new file when it is closed, thereby maintaining files in consecutive sequences of blocks. The 'close' operating system call could invoke a background process that checks to see if the file is contiguous, and if not, then checks to see if enough contiguous blocks are available. If so, the background process could allocate the free blocks, copy the file over, free its old blocks, and re-sort the free list.

A possible disadvantage of this strategy would be a perceived degradation in performance by other users, because mutual exclusion would be required while manipulating the list of free blocks, but this would probably be more than offset by the resulting overall improvement in performance.

Larger Block Size

Increasing the block size is an important optimization in the Berkeley file system, and deserves attention. As mentioned above, it offers an inherent reduction in fragmentation, is a form of read ahead, and saves on indirection, but it wastes space. Since increasing the block size is only practical if it does not waste space, some space-saving strategies are considered here.

Space Saving Strategies

The Berkeley System of Fragments

The Berkeley system of allowing files to occupy fragments of a block is complex, as befits an academic institution, but that may not be very important since they have tested the design and it can be imitated. It apparently solves the problem of increasing the block size without wasting space.

Allocating Blocks in Clusters

Another way of achieving a large effective block size is to allocate several blocks at a time, and have the kernel do all of its I/O in chunks of several blocks.

So far we have not saved any space, but it should be relatively simple to devise a way to free the unused odd blocks and use them for small files or the ends of large files. Some dynamic reorganization would be necessary to copy small files into odd blocks, and to copy files out of odd blocks if they outgrow them.

On the face of it this seems simpler than the Berkeley fragmenting strategy. The next step would be to do a more detailed design and see if it is still simpler.

More Intelligent Buffering

The objective of the buffer system is to save data that may be needed again, so it may be worthwhile to try to predict which data is most likely to be needed. The current kernel uses a "least recently used" (LRU) algorithm which is implemented very simply by maintaining a free list of buffers in that order. We could consider more sophisticated general algorithms such as "least frequently used," but this is probably the road toward greatest possible complexity.

Aside from general algorithmic approaches, we can try to distinguish between types of data that may be needed more frequently. We can distinguish between the contents of files and the file system constructions such as inodes, directories, and indirect blocks.

Directories are needed only for opening, creating, or deleting files, which is to say not very often, but on the other hand, traversing path names might involve searching the same few directories over and over. This suggests that it might be useful to cache the parent directories of the current directory.

Inodes are stored in memory independently of the buffer system, but they must be updated if a file is being written to and they must also be time stamped, so that buffering them might be worthwhile.

Indirect blocks might also be buffered. For a file large enough to need indirect blocks, the kernel must refer to them in order to find the data blocks of the file, and when writing to a large file, the kernel must update the indirect blocks. The rationale for indirect blocks was that the user must pay a price for large files.

Conclusions

All of the strategies are closely interrelated. For example, if we do program loading and large raw I/O requests by DMA to or from the user memory, we avoid the overhead of buffering these requests block by block, and we also save our buffer cache from getting flushed out. However, there are also some other implications. First of all, this would reduce the amount of sequential I/O, as opposed to random I/O, passing through the buffer system, and thus reduce the need for read-ahead algorithms. Secondly, it would put a premium on maintaining files in contiguous blocks on the disk, so that large I/O requests could proceed at the data transfer rate of the I/O devices. If files are maintained contiguously, then it might make sense to use a track-cacheing disk controller, where otherwise it would not. Thirdly, by reducing the amount of sequential I/O passing through the buffers, it might make buffer cacheing more effective by preserving more frequently used blocks in the buffers. Thus certain optimization strategies go together and compound each other's effects, reducing or eliminating the need for other strategies, and these in turn influence the most desirable hardware architectures.

Our plan for future investigation at Intel is to try as many of the optimizations mentioned in this article as possible, in order to find out which ones and which combinations are most effective for various types of applications and system loads.

References

McKusick, M., Joy, W., Leffler, S., and Fabry, R., "A Fast File System for Unix," Computer Systems Research Group, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, CA.

/usr/group Standards Effort

Heinz Lycklama
Chairman, /usr/group Standards Committee

/usr/group Standards Effort
UNIFORUM
January, 1984

Heinz Lycklama

Chairman,
/usr/group Standards Committee

1. Introduction

The /usr/group Standards Committee was formed to formulate, adopt, publish, and provide a formal standard specification, based on the UNIX* operating system developed by Bell Laboratories, for a commercial operating system. The purpose, benefits, and format of the Standards document are discussed. This document is intended to give the reader an appreciation for the scope of this effort. Some of the history of the commercial UNIX system market is covered to give the reader a historical perspective.

The /usr/group Standards Committee was formed in the summer of 1981 by the Board of /usr/group in response to the need for a standard in the commercial UNIX marketplace. The existence of a standard will benefit both producers and consumers of UNIX-based software. The Committee has put great emphasis on establishing the proper organization for this effort so that the proper procedures are in place when the time comes to vote on a standard. The Committee is sensitive to legal and competitive issues involved with attempting to establish a standard. Efforts are currently being focused on identifying the set of system calls and subroutines which will comprise the system interface standard and on identifying a set of extensions to the system interface which are frequently required by commercial applications.

The benefits of having such a standard are enormous in the exploding UNIX applications marketplace. Interchange of programs, data and personnel is greatly facilitated. The educational and economic benefits are self-evident.

The committee is composed of about 40 key representatives from the UNIX-based system, UNIX-like system and UNIX applications vendors, as well as from the community of UNIX users. The recent addition of representatives from Bell

* UNIX is a trademark of Bell Laboratories.

Laboratories has strengthened the ability of the Committee to define a standard on which a large section of the UNIX community can agree.

2. Versions of UNIX

The UNIX system was developed at Bell Telephone Laboratories in the early 1970's to provide a convenient software development system. It was originally written for the PDP11 computer in assembly language. One of the most significant developments in this effort has been the fact that the complete system was rewritten in the low-level systems implementation language, C, for the PDP-11/45 computer in 1973. In 1974, Version 5 of the UNIX system was released to educational institutions. It gained popularity very rapidly. This led to its availability in the commercial market in 1976. In 1977, a number of companies started to provide support for the UNIX system to commercial sites. The next version of the UNIX system, PWB UNIX, was released in 1977 as well. However, the high costs of source licenses and binary licenses prohibited the wide adoption of UNIX systems at commercial sites. In the meantime, the availability of UNIX licenses to educational institutions for a nominal distribution fee encouraged the widespread use of the UNIX system at universities. Later versions of the UNIX system (Version 7 and System III) had lower binary prices and less restrictive licensing, thus increasing the popularity of the UNIX system in the commercial world.

One of the main stimulants to the standardization efforts has been the fact that there are now a large number of versions of UNIX and UNIX-like systems being marketed commercially. This includes versions of the UNIX system as distributed by AT&T, UNIX-based systems marketed by some commercial vendors, licensed by AT&T, and UNIX-like systems marketed by companies without a license from AT&T. This poses a real problem for those who are attempting to develop applications for the UNIX marketplace. What systems interface do the applications vendors program to?

There are a number of versions of the UNIX operating system available from AT&T. These include V6, PWB, V7, 32V, System III and System V. System V Release 2.0 was announced in January, 1984. The later versions of the UNIX operating system are distributed for both the VAX and the PDP11 computers. Internally, Bell Laboratories has standardized on System III, and subsequently, on System V. The intent is to make later versions of the UNIX system available to commercial vendors at the same time as internally to Bell. AT&T has also made a commitment to make later versions of the UNIX system upward compatible with previous releases, starting with System III.

The majority of the UNIX systems vendors market UNIX-based operating systems. That is, they have modified and/or extended the original UNIX operating system, as released by AT&T and remarketed the system under their own names. Most of the recent systems are based on UNIX System III, and a few on System V. One of the predominant exceptions to this is the Berkeley 4.1BSD system for the VAX computers, which is based on 32V, which in turn is based on UNIX Version 7. More recently, the Berkeley 4.2BSD system has been released. The common trade names used for the more popular UNIX-based operating systems include IS/3, XENIX, UNIPLUS, UNITY, VENIX, Zeus, ONIX, UTS, etc.

Then there are the UNIX-like operating systems, which the vendors claim are compatible with the UNIX system in anywhere from spirit to complete specifications and utilities. The IDRIS system, marketed by Whitesmiths, claims to be largely V6 compatible. The UNOS system, marketed by Charles River Data Systems, and the Coherent system, marketed by Mark Williams, claim to be largely compatible with V7 UNIX, with some extensions of their own. Other UNIX-like systems have also been announced recently.

With this diverse set of operating systems, the need for standardization arises.

3. History

Early in 1981, the /usr/group Board of Directors determined that a formal standardization effort could be of significant benefit to the general membership of /usr/group. Heinz Lycklama was appointed to serve as Chairman of this standards effort and to organize a Committee that would propose, define and publicize this standard. The first organizational meeting was held in conjunction with the /usr/group meeting held in Los Angeles during July of 1981. About forty-five people expressed interest in working on the /usr/group Standards Committee. A sense of the charter of the committee was developed at this meeting. The intent was to provide a broad representation of systems vendors, applications builders, and end-users in the UNIX marketplace on the committee. A Steering Committee of eight (8) members, including the Chairman was chosen to guide the efforts of the standards group. In addition, about 30 other members were chosen to act in an advisory role in the Committee. Subsequent to this a number of other Committee meetings were held in conjunction with national UNIX-related conferences. These include:

December 1981 at /usr/group meeting in Boston
July 1982 at UNICOM conference in Boston
August 1982 Systems Interface Sub-Committee meeting in Chicago
November 1982 at COMDEX in Las Vegas
January 1983 at UNICOM in San Diego
July 1983 at UNICOM in Toronto
November 1983 at COMDEX in Las Vegas
January 1984 at UNIFORUM in Washington, DC

The major part of each of these meetings has been devoted to defining the System Interface Standard, referred to as the /usr/group Standard.

4. Benefits

The potential benefits to all persons using the UNIX or functionally compatible operating systems are considerable for the following reasons:

1) Consumers will have a wider variety of products to choose from within the context of their own specific applications areas.

2) Developers will be able to create products that can be sold in a much larger marketplace.

3) Suppliers will enjoy an increased demand for their products because of a wider variety of applications making use of these products.

4) Employers will find it easier to hire experienced, technically proficient staff; and the cost of training new or existing employees should be significantly reduced.

5) Employees will enjoy greater job security because of a larger market for their skills and the reduced likelihood of technical obsolescence.

6) Students will find the transition from theory to practice easier; instructors will find a larger selection of instructional materials available; and researchers will find it easier to exchange ideas.

The economic and educational benefits of the standard are well worth the effort of defining a standard at this time.

5. Purpose

In the light of the considerable potential benefits to all members of /usr/group, the Standards Committee has spent a considerable amount of time trying to clearly define its purpose. This statement now reads:

> "The purpose of the /usr/group Standards Committee is to formulate, adopt, publish and promote a formal standards specification, based on the UNIX operating system, for a commercial operating system. This standards specification is intended to assist persons producing or acquiring products based on the UNIX or functionally-compatible operating systems in accurately predicting the behavior of the products in the context of a specific implementation."

6. Organization

The Committee determined at its initial meeting in December of 1981 that several issues were of immediate importance to the standards effort. Consequently, one temporary and three permanent Sub-Committees were formed so that the Committee could pursue the following goals simultaneously:

1) To identify the set of system calls and subroutines which will comprise the initial system interface standard; and to resolve any ambiguities which are contained in the present definitions of these functions.

2) To identify a set of extensions to the system interface which are frequently required by commercial applications; and to define the operational characteristics of these functions.

3) To locate and/or define a viable standard for the "C" Programming Language.

4) To draft an organizational charter which outlines the purpose, benefits, costs, and feasibility of developing a viable commercial standard; and to draft a set of procedures by which the Standards Committee may conduct its proceedings and by which the /usr/group may formally adopt the standards developed by the Standards Committee.

The organization currently consists of eight Steering Committee members and about 30 advisory Committee members. The following are currently members of the Steering Committee.

John L. Bass	Member
David L. Buck	System Interface/Extensions Sub-Committee Chairman
Richard Hammons	Member
Tom Hoffman	Secretary/Treasurer
Don Kretsch	Member
Heinz Lycklama	Chairman
Jeff Schriebman	Member
Bob Swartz	Member

Steering Committee members have been chosen to insure a broad representation of producers, integrators, and end-users of the products which the /usr/group Standard is intended to benefit.

Advisory Committee members have been chosen based upon their expression of interest in the standardization activities. The current members of the Advisory Committee are:

J. Boykin
B. Boyle
P. Caruthers
C. Cole
A. Cornish
W. Corwin
D. Cragun
I. Darwin
L. Ford
C. Forney
M. Gien
S. Glaser
T. Godino
J. Goldberg
J. Isaak
B. Joy
M. Katz
D. Ladermann
G. Laney
M. Laschkewitsch
B. Laws
J. McGinness
E. Petersen
P. Plauger
T. Plum
H. Stenn
T. Tabloski
M. Teller
M. Tilson
D. Wollner

At various times, there have been four active Sub-Committees: (1) Systems Interface, (2) Extensions, (3) C Language, and the (4) Organization and Procedures Sub-Committees. Most of the activity is now concentrated in the combined System Interface and Extensions Sub-Committee. However, a Validation Sub-Committee has recently been formed to look into validation suites.

6.1. System Interface/Extensions Sub-Committee

The initial system interface standard is based on UNIX System III and attempts to maintain compatibility with UNIX Version 7 wherever possible. The standard is intended to provide a basis for commercial applications development, and is an attempt to define a system interface based on the UNIX operating system. Special attention has been given to reducing or eliminating particularly machine-dependent functions.

Effort has concentrated on the original Sections 2 and 3 of the UNIX System III User's Manual. Section 2 defines the system calls which comprise the basic interface to the operating system. Section 3 defines the subroutines, both intrinsic and system call specific, that define the programming environment. Functions were omitted from the initial standard primarily because they either exhibited a high degree of machine dependency or were not usually required for applications development in the commercial marketplace. The Sub-Committee is presently editing manual sections based on UNIX System III and hopes to have a proposed standard ready for consideration by the general membership early in 1984.

The first major topic that was considered by the Extensions

Sub-Committee was simply to determine what constituted an extension. After some discussion, it was generally agreed that extensions could include any commercially useful functions or facilities which were not described in the documentation distributed by Bell Laboratories, Western Electric, or AT&T. Extensions could, therefore, include any system calls, subroutines, utilities, or other programs which are not specifically documented in Volumes 1 and 2 of the UNIX Programmer's Manual.

Prior to the formation of the Extensions Sub-Committee, the Standards Committee as a whole had determined that the single most important technical subject that needed to be dealt with was the problem of controlling contention between multiple concurrent processes for shared data files ("record locking"). Three written proposals were considered from Dwayne Peachey of the Hospital Systems Study Group, John Bass of Fortune Systems, and E.J. McCauley of Zilog. After reviewing the various characteristics of each of these three proposals, the Sub-Committee decided to recommend adoption of the technique proposed by John Bass. Upon the Sub-Committee's recommendation, Mr. Bass agreed to clarify portions of the user documentation and to provide a brief explanation of the technical rationale behind some of the more subtle characteristics of the technique. The description of this technique, including a suggested means of implementation, should be available for distribution to the /usr/group general membership early this year.

The Sub-Committee is actively soliciting written proposals from the /usr/group membership. Individuals wishing to make Extension proposals to the System Interface/Extensions Sub-Committee are encouraged to become members of the Standards Committee in order that they may present and explain their proposals in person. Individuals who are not members of the Standards Committee must find a member of the Committee to sponsor their proposal.

Written proposals should address issues of general interest to the /usr/group members and should document techniques or facilities which have already been implemented. It is preferred that the source code for implementing the technique be in the public domain or the owner of its copyright must grant permission to the /usr/group Standards Committee to reproduce and distribute copies of the technique. Documentation or source code protected as proprietary information or trade secrets will not be accepted by the Standards Committee for consideration.

6.2. "C" Language Sub-Committee

In December of 1981, it was learned that Bell Laboratories was planning to initiate an ANSI standardization effort for the "C" Programming Language. In March of 1982, Bell

Laboratories proceeded with this effort. In light of this, it was decided that the focus of the "C" Language Sub-Committee should be to maintain close contact with this ANSI standardization effort and to take an active role in the preparation and presentation of this standard to ANSI. The Standards Committee maintains an active interest in this area.

6.3. Validation Sub-Committee

This Sub-Committee was formed to investigate the availability of validation suites that could be used to validate systems that conform to the /usr/group Standard. While the committee cannot take an active role in validating systems, it will probably be possible to allow parties not associated with /usr/group to offer their services in this area. The details are being worked on by the Sub-Committee.

7. Committee Organization and Procedures

The organization and procedures temporary Sub-Committee developed the organizational Charter and By-Laws. The revised documents have been distributed to all members of the Standards Committee and the /usr/group Board of Directors, and approved.

7.1. Charter

The organizational Charter outlines the purpose, goals, scope, benefits, costs and feasibility of developing and implementing a viable standard. The Charter was written according to the format suggested by ANSI to facilitate a possible future submission of the standard to this body.

7.2. By-Laws

The By-Laws describe the structure and procedures of the Standards Committee itself. Briefly, the Committee consists of eight Steering Committee members and up to thirty-two Advisory Committee members. Members of either Committee must be a general member of /usr/group.

7.3. Steering Committee

Steering Committee members serve two-year staggered terms and are selected from the Advisory Committee by the Standards Committee Chairman, subject to the approval of other Steering Committee members. The Steering Committee is responsible for determining topics of discussion and for controlling publication of all standards documents.

7.4. Advisory Committee

The Advisory Committee meets twice each year, at least, in conjunction with the regular /usr/group meetings. The primary criteria for membership in the Advisory Committee is interest, as evidenced by active participation in Standards Committee activities. Advisory Committee members may suggest topics for discussion and submit written proposals for inclusion in the standard. All draft standards will be published and distributed to members of the Advisory Committee for review and comment prior to being distributed to the /usr/group general membership. Advisory Committee members are nominated by the Chairman and approved by the Steering Committee.

8. Format

A long-term goal of the Standards Committee is to present the completed standards specification to the International Standards Organization (ISO) and the American National Standards Institute (ANSI) for adoption as an official ISO and/or ANSI Standard. Thus the format of the document closely resembles others submitted to ANSI for adoption as a Standard.

The final format of the Standard includes several sub-parts, some of which may be optional. The following guidelines were used in editing the Draft Standards document, which is based on the UNIX System III User's Manual.

a) The material is presented in a style familiar to the UNIX programmer, but clearly identifiable as the /usr/group Standard.

b) All machine-dependent references and facilities are removed.

c) Many octal constants are translated into symbolic equivalents, using existing symbolic constants wherever possible.

d) All constant values which imply limitations of magnitude and are implementation dependent are turned into separately identifiable symbolic constants, defined in a separate Appendix.

e) All references to non-existent parts of the Standards document are removed.

f) System call and subroutine names which are not part of the Standard are put on a reserved name list.

g) The Standard will be published along with a set of limits which must be specified by any Standard-conforming system.

The Standards document which is to be distributed for voting purposes actually come in two parts. The first part is the /usr/group Standard document itself. It consists of some introductory material plus edited manual pages derived from sections 2 and 3 of the UNIX System III User's Manual. The first section of the manual is essentially kept as a place holder for possible future standards efforts. Section 4 is also kept as a place holder. The other sections from the original UNIX documentation are combined into one section labelled as 'Miscellaneous'. This section contains only the edited manual pages required as referenced by manual pages in sections 2 and 3.

The second part of the document is a reviewer's guide. This is provided, as the name indicates, to provide the reviewer a roadmap to the /usr/group Standard document itself. The guide highlights the manual pages which have undergone major editing. Differences between the UNIX System III routines and the /usr/group Standard routines are pointed out as a guide for the reviewer. It also contains a list of system calls and subroutines that comprise the /usr/group Standard system interface. These are listed alongside the list of system calls and subroutines that appear in System V, System III, Version 7 and 4.1BSD versions of the UNIX system for comparison purposes.

9. Procedures

The procedure for adopting a standard consists of the following five basic steps:

1) A written proposal is submitted to or developed by the Steering Committee.

2) The Steering Committee Approves publication of a "DRAFT" standard which is then distributed to the Advisory Committee for review and comment.

3) The Steering Committee incorporates these comments into the standard and approves publication of a "DRAFT" standard which is then distributed to all general members of /usr/group for review and comment.

4) The Steering Committee incorporates these comments into the standard and approves publication of a "PROPOSED" standard, which is then distributed to all general members of /usr/group for voting by written ballot.

5) If two-thirds of the general members responding with written ballots are in favor of the "PROPOSED" standard, it becomes an "ADOPTED" standard and part of the official /usr/group Standard.

In summary, the process of adopting the standard involves soliciting comments from a group of individuals, revising the standard based on these comments, and distributing the revised standard to a larger group of individuals for further comment. Once a high degree of consensus has been achieved, a "PROPOSED" standard is published and distributed to the general membership of /usr/group for voting. If two thirds of the ballots returned are in favor of the proposed standard, it is considered formally adopted.

10. Schedule

The current schedule for publishing the Standards Document is as follows:

Nov 30, 82	COMDEX Meeting to discuss comments from Standards Committee and vendors.
Jan 25, 83	UNICOM Meeting to review DRAFT standards document.
Feb 28, 83	DRAFT Standard Document made available to /usr/group office for distribution to interested parties for comments.
Jul 6, 83	Last day for receiving comments on the DRAFT Standard document.
Jul 15, 83	Review of comments on DRAFT Standard document by Standards Committee
Nov 29, 83	Review of Toronto changes by Standards Committee
Dec 23, 83	Publish PROPOSED Standard for Standards Committee review.
Jan 17, 84	Final review of PROPOSED Standard by Standards Committee
Mar 6, 84	Publish PROPOSED Standard for /usr/group general membership vote.
May 4, 84	Deadline for receiving Ballots on PROPOSED Standard.
Jun 4, 84	ADOPTED /usr/group Standard ready for distribution.

However, meeting this schedule assumes that the reviews are largely favorable and that there are no publication snags.

11. Legal Issues

The three most important legal issues that the Standards Committee continues to be concerned about involve copyright, trademark and antitrust law.

11.1. Copyright Issues

The material that we are using for editing the /usr/group Standard is the UNIX User's Manual for System III. This is copyrighted by Bell Telephone Laboratories. We have permission from AT&T to use selected pages from this manual for inclusion in documents published by /usr/group provided that the source of any such pages is acknowledged and any relevant copyright notice is reproduced.

11.2. Trademark Issues

UNIX is a trademark of Bell Laboratories. To avoid problems with AT&T we should use the "TM" trademark symbol after each occurrence of the word "UNIX". Any reference to the UNIX system must be a reference only to a system developed by Bell Laboratories and furnished pursuant to a license with Western Electric or AT&T. Any system furnished by an AT&T licensee cannot be called a UNIX system, but only derived from a UNIX system. The word "UNIX" can only be used to identify operating systems or other software whose source is Bell Laboratories. The Standards Committee must avoid using the word "UNIX" when referring to the /usr/group standard interface.

11.3. Antitrust and Unfair Competition Issue

The Standards Committee has taken precautions to avoid claims that use of the standard violates antitrust laws or the common law of unfair competition. There is no attempt to stabilize or fix the price of operating systems, to eliminate competition or to control production levels. The Standard is being adopted on a suggested basis and /usr/group members are not required to adhere to the /usr/group standard interface. /usr/group will not disparage or sanction a member or non-member who makes products which do not conform to the standard interface.

An attempt is being made to give all persons affected by the Standard, whether they be vendors, users or OEM's, an opportunity to participate in the process of creating the Standard. As another precaution, /usr/group will not test products to determine the degree to which they conform to the proposed Standard.

12. Status of the /usr/group Standard

At the last committee meeting, held in Washington, D. C. on January 17, 1984, the /usr/group Standards committee passed a motion, by a vote of 24 to 3, to submit the Standards document to the general /usr/group membership for a vote on the adoption of the Standard as it now stands.

After a final round of editing and phototypesetting, the

Standards document, along with the Reviewer's Guide, is being sent to the /usr/group membership for a vote. This is expected to happen towards the beginning of March, 1984. A two-thirds vote is required for adoption of the Proposed Standard. The documentation will include a comparison with System III and System V, as well as a detailed description of the file/record locking primitive so that members can vote with access to the proper background information.

In further discussions held on January 17, extensions in the areas of terminal handling, networking, real time, interprocess communication and object file formats were considered. It was felt that something had to be done about the IOCTL system call so that the terminal control capabilities could be specified. After some debate, it was agreed that the System V version of IOCTL would be the best starting point. A proposal will be generated for consideration at the next /usr/group Standards Committee meeting, to be held in April in conjunction with the Winter Comdex meeting in Los Angeles.

Experimental Implementation of UUCP — Security Aspects

D.A. Nowitz
AT&T Bell Laboratories
Murray Hill, NJ 07974

P. Honeyman
Princeton University
Princeton, NJ 08554

B.E. Redman
Central Services Organization
Whippany, NJ 07891

Experimental Implementation of UUCP — Security Aspects—

D. A. Nowitz

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

P. Honeyman

Princeton University
Princeton New Jersey, 08544

B. E. Redman

Central Services Organization
Whippany, New Jersey 07981

ABSTRACT

The *uucp* network software has been running for over 5 years. During that time, a number of bugs were uncovered and many enhancements were suggested. Early this year, a group of interested *uucp* administrators met to discuss the production of yet another improved version of *uucp*. As a result, three people divided up the work and produced an experimental version with the following major enhancements:

- The security aspect was reworked to provide more restricted file access, to provide a clearer mechanism for specifying these permissions, and to provide easier initial setup.
- A more flexible system for specifying and implementing various connection media is implemented; for example, Develcon and Micom switches, and Ventel dialers.
- The spool directory is now implemented as multiple directories; this improves work search time and prevents blocking due to faulty communications to a particular system.

In addition, effort was applied to simplify the code, improve the algorithms, and provide additional administration tools. This paper will focus on the security aspects of the new software.

It is now possible to specify that some remote sites can only send files, they can not request files; this provides a high degree of security while letting remotes communicate for purposes such as mail. In addition, another option can further restrict communications by not permitting the transfer of queued local requests during a conversation initiated by a remote site. With this option, the local secure site must call the remote in order to transfer queued files; a masquerader can not call up and receive files destined for someone else.

The directories that can be accessed for reading or writing by a remote machine are specified by *READ* and *WRITE* options. Commands that are executed by *uuxqt* can be specified on a machine by machine basis. For machines that have special execution privileges, a *VALIDATE* option is available to check the login-id against the machine name.

All the options are specified in the *Permissions* file. The defaults are for maximum protection; options must be specified to give greater freedom. An attempt was made to make it easier to read and understand this scheme; the entries in the file look like shell or makefile variables, for example LOGNAME=nuucp. In addition, a program is available that reads the *Permissions* file and prints an English version of how the *uucp* programs will interpret the file.

Introduction

"Your *USERFILE* is set up wrong!" Over the last 5 years, I've used that line many times. And it's no wonder; whenever the problem came up, it was always easier to give the fix, or fix it myself rather than explain how it was suppose to work. As a result, many say, "uucp was not designed with security in mind." Here are several factors contributing to that assessment—there are others:

- The defaults for the system were for few restrictions unless overt action is taken.
- There were bugs in the file access code, undetected for several years.
- It is difficult to explain how to set up the *USERFILE* to provide file access restrictions.
- The spooled files were generally readable by everyone.

Over the years, thought has been given to these things, and when the group started forming to produce a better *uucp*, we thought about them some more. The challenge was to strengthen security within the present framework. The result is the subject of this paper—the security aspects of the experimental *uucp*.

In this version we provide the correct defaults when the administrator doesn't say anything, provide more flexible restriction options, and provide a program that tells the administrator how the *uucp* programs interpret what was specified.

Parts of this paper assume knowledge and understanding of the current version of *uucp* and its administration.

Who's Who

Throughout this paper, reference will be made to remote machines—those machines that my machine will call or that will call my machine, and the local machine—the machine that I administer. For simplicity we will refer to the remote machine using second and third person pronouns and use first person pronouns for the local machine.

Notice that the protection mechanisms described in this paper refer to remote users; local users can queue up and send any files they have access to read. No attempt is made to classify or restrict local users.

When Someone Calls Me

In *uucp*, as in the UNIX* system, the first line of defense is the login process. If someone obtains the login/password information of a local user, there is little further security protection available. However, with *uucp*, having the login/password information is less useful, since a standard shell is not provided for that login. Rather, the *uucico* program is invoked upon logging in. Given these facts, we designed the file access permissions for remote sites to log into my system to rely entirely on the login-id.

One of the two types of entries in the *Permissions* file is the *LOGNAME* entry. (The other is the *MACHINE* entry). An administrator must provide at least one *LOGNAME* entry in the *Permissions* file so that a remote site can call in and start a conversation. For example,

```
LOGNAME=nuucp
```

states that you can login as "nuucp" and since no other permissions are specified, all the default permissions/restrictions are provided:

- You can send files exclusively to the *uucp public* directory. (usually /usr/spool/uucppublic)
- You can *NOT* request to receive any files.
- Files that are queued for you will *NOT* be transferred during the present session.

* UNIX is a trademark of AT&T Bell Laboratories.

- The only commands that you can execute are the defaults—usually "rmail" and "rnews".

The administrator can now provide less restriction, hopefully because the login-id and password are more tightly controlled—friendly systems.

```
LOGNAME=uucpa   SENDFILES=yes
```

I believe you are who you say you are, so I will send the files that are queued for you on my system after you finish sending your files to me. (This in effect, permits the *uucico* program to switch from *SLAVE* to *MASTER* mode.)

The administrator can further open the system to *friendly* systems with

```
LOGNAME=uucpa   REQUEST=yes
```

You now have permission to request files from my system, but the pathname is still restricted as above. The *SENDFILES* and *REQUEST* options are independent and can be used together.

Now we'll turn to file access permissions. The default read and write access is to the *uucp public* directory—/usr/spool/uucppublic/. Additional (or more restrictive) permission can be specified using the *READ*, *WRITE*, *NOREAD*, and *NOWRITE* options. Each is a colon separated list of full path names, for example,

```
WRITE=/usr/spool/uucppublic:/usr/news
```

gives access to the /usr/news directory in addition to the *uucp public* directory tree.

```
WRITE=/dev/null
```

has the expected result. But note that mail is still permitted, because internal (within the *uucp* system) transfers to the spool directory are permitted.

A really open system might have

```
READ=/   NOREAD=/etc
```

allowing files to be read from anywhere except the /etc directory tree. Notice that in order for reading to be permitted at all, the *REQUEST* option must be specified with value "yes", since not specifying the option defaults to "no"—you can not read any files from my system.

To summarize, the minimum entry for remotes to login is

```
LOGNAME=nuucp
```

and the most permissive entry is

```
LOGNAME=uucpa   SENDFILES=yes   REQUEST=yes \
READ=/   WRITE=/
```

Again note that for file transfer who you are is not relevant, login-id is all that counts.

You may want to be sure of the caller, and don't care about the phone bills. In that case, you can use

```
LOGNAME=uucpc CALLBACK=yes
```

This will cause the conversation to terminate and an attempt to call the remote site will occur. In addition, a dummy "C." file is created in the spool directory; this to ensure that a call will be made if the first attempt fails.

When I Call Someone

Now that the previous section is well understood, this section is fairly simple. The *MACHINE* entries are used to provide permissions to systems when the local calls them. The *REQUEST*, *READ*, *WRITE*, *NOREAD*, and *NOWRITE* options are exactly the same and have the same defaults as the *LOGNAME* entries. The *SENDFILES* entry has no meaning, since I called you to send you the queued files. One other difference is that no *MACHINE* entries are required in the *Permissions* file; any system I call whose name does not appear in a *MACHINE* entry will get the defaults:

- My send and receive requests will be executed.
- You can send files to my system's *uucp public* directory.
- Your commands that I will execute are in the default set. To specify anything different, a *MACHINE* entry must be specified for the machine in question. Note that both the *LOGNAME* and *MACHINE* entries can both take values that are colon separated lists, for example

```
MACHINE=eagle:owl:raven:hawk:dove  \
 REQUEST=yes  \
 READ=/  WRITE=/
```

specifies that all five birds can do almost anything they want as far as file transfer is concerned.

Sharing

Because there are different entries that can take effect when the same two systems are talking, one when I call you and one when you call me, some non-deterministic results (from the user's point of view) can occur. The user can attempt to send some files to a friend's directory, and sometimes they will be rejected while other times they will be accepted—depending on which machine initiated the call. To alleviate this situation and to allow smaller *Permissions* files, *MACHINE* and *LOGNAME* entries can be combined. For example,

```
LOGNAME=uucpz:uucpa  \
MACHINE=eagle:owl:raven:hawk:dove  \
 SENDFILES=yes  REQUEST=yes  \
 READ=/  WRITE=/
```

Execute Please

There is a major difficulty in granting different execute permissions to different systems. The execution demon, *uuxqt* is run after a conversation is complete, so it doesn't know what machine sent the command. As pointed out earlier, for file transfer, the login-id can be used to distinguish between classes of users. But for command execution, the information must be made available to *uuxqt*. In addition, when I call another machine, the identity of that machine must be made available to *uuxqt*.

What is required, then, are two things: giving *uuxqt* a way of knowing what machine sent the command, and some way to validate who the remote is when additional command permission is allowed. Due to an auspicious design decision we made, one of these comes for free! We decided to hash the spool directory based on machine name—each machine that I talk with has its own spool directory with its name as the name of the directory. When *uuxqt* executes, it can base the command permissions on the name of the directory where the "X." file resides.

Great! All that is left to do is guarantee that you can't put something in someone else's spool directory and have a way of preventing masquerading by remote sites. The first is already taken care of with the defaults of file access and the *WRITE* option. And although we can't guarantee the second, we can provide the same level of control as with any other UNIX login/password protection.

There is little problem with one half of the guarantee; when I call you, I'm fairly confident. (I could have the wrong number, but these sites are usually well known to me so knowledge of phone number changes will be readily available). Also, someone could have gotten access to your PBX and re-routed your calls—this will have to be addressed in some future version). So the problem boils down to some type of validation when you call me.

The *VALIDATE* option is available for *LOGNAME* entries and has the following property: Any time you login and identify yourself as a machine that appears in a *VALIDATE* option, your login-id must correspond to one of the *LOGNAME* names. For example,

```
LOGNAME=uucpa:uucpb  \
  VALIDATE=eagle:owl:raven:hawk:dove
```

specifies that if you call me and say you are "eagle", then the current login-id must be either "uucpa" or "uucpb". If not, I'll terminate the conversation. If the administrator protects the "uucpa" and "uucpb" logins the same way as any other login for the local system—only give it to trusted people/machines—then I have the same level of protection as available to other users' logins. Granted, the fact that the data is on another machine creates some hazard. However, the situation for allowing "unsafe" commands is usually between machines that are locally administered.

Now that there is some confidence about whom I'm talking to when I will be granting special execution permissions, I can actually grant the permissions:

```
MACHINE=eagle:owl:raven:hawk:dove   \
  COMMANDS=rmail:rnews:who:ls:lp:diff
```

The *COMMANDS* option is used with the other *MACHINE* options, but is the only one used by *uuxqt* exclusively.

Conclusion

We have described some to the security features of the experimental version of *uucp*; we have not given a complete description, but rather a feeling of how and why it was designed and implemented. In our design, we tried to make it easier to explain, easier to read and interpret the data file, and provide for restriction by default with overt granting of permissions. (We also provide a program that reads the file and prints an English like interpretation of how the *uucp* programs will interpret the specific *Permissions* file).

Here are a few areas for further protection investigation: the use of public key encryption for machine validation, using an encryption scheme for internal data transmission and spool directory storage, and providing gateway facilities.

Mapping the UUCP Network

Rob Kolstad
Convex Computer Corporation
1819 Firman #151
Richardson, TX 75081

Karen Summers-Horton
2843 Valcour Court
Reynoldsberg, OH 43068

MAPPING THE UUCP NETWORK

Rob Kolstad

CONVEX Computer Corporation

Karen Summers-Horton

The UUCP network encompasses those machines on the UNIX News Network (Usenet) and more: over 2100 sites as of January 11, 1984. This talk details the difference between the UUCP network and Usenet in addition to outlining the current problems in using the networks (notably those concerning reliable routing among sites: "how to get there from here"). If the UUCP network is to become an ARPA domain, reliable maps and site descriptions must be available to a "name server". This talk will also explain our current plans for mapping not only the connectivity of the network but also the "quality" of the connections for use with routing programs such as Steve Bellovin's optimal path finder. The talk will include current schema for collecting, disseminating, and using the information.

1. UUCP and Usenet

UUCP (Unix to Unix CoPy) is a file transport mechanism which also features the ability to request execution of a few commands at remote sites (e.g., rmail and rnews). It is used primarily to send electronic mail and news. The network is not always "connected": a host often calls another site only on a schedule or when traffic is waiting for the remote site. AT&T Bell Laboratories designed UUCP in 1976 with the assumption that all machines had an auto-dialer and could call all other machines directly (a valid assumption when there were only a few hosts running UUCP). Nowadays, each host connects to one or several remote machines. These hosts range in size from small microcomputers (e.g., Radio Shack TRS-80 model 16's) to giant multi-user systems.

UUCP and rmail use a clever idea to move mail to sites not directly connected to a host. The program "rmail" examines a mail header (which specifies sites connecting the original host and the mail's destination) and sends the mail along the first site listed on the list after discarding it. To send mail from site 'parsec' to user 'joe' at site 'adec23', a path like this emerges:

```
allegra!alberta!sask!hssg40!adec23!joe
```

Clearly, it is of utmost importance to know who talks to whom! These paths become more complicated when internetwork routes enter the picture. It is possible to send mail from the UUCP network to the ARPANET, DEC E-NET, BITNET, and many local networks. Today, over 2100 hosts run UUCP; this is where our problems begin.

A fraction of these machines (approximately one-third) also belong to a logical subnetwork called "Usenet". Usenet is an electronic bulletin board connecting about 700 sites in the United States, Canada, Europe, and Australia. A person at any Usenet site may post articles to any specific bulletin board (called a "newsgroup") and (eventually) reach all people who subscribe to that newsgroup.

News is transferred by various means: UUCP, local area networks, and long haul networks to name a few. Maintenance and development of Usenet software is done on a volunteer basis; the cost for transferring information is borne by each individual site. A Texas site pay $2,000 to $4,000/year to the phone company for long distance charges for news and mail. Some sites pay nothing: Southern Methodist University (with no budget) is polled by local industries. Digital Equipment Corporation maintains some overseas links; their phone bill is around $19,000 per month.

2. Problems with UUCP

Along with the advantages of low cost and ease of use, the UUCP network has problems. In summary, these problems are:

a. Explosive growth

b. Addressing reliability

c. Lack of backward error recovery

d. Unknown and unpredictable propagation delays

We will address the problems of growth and addressing here.

UUCP is growing by leaps and bounds. Our current guess of over 2100 hosts is just that: a guess. There is no official central registry of UUCP hosts. When someone joins UUCP, there is no official procedure. Once they find someone who will let them hook up to their site, they then can mail to the whole net without anyone except their immediate neighbor really knowing who they are. As a result, trying to get mail to John Smith at Framus, Inc. is difficult unless you happen to know the path through the network to the Framus computer. With over 2100 sites, (most of which are not directly connected to each other), this becomes nearly impossible. To find your way through the net, you need a road map. Our project is to provide that map.

Since the net is growing so quickly and since it is so easy to set up or tear down a link, the net's structure is never fixed. Not only must the initial map be collected, but it must be constantly maintained or it will be quickly out of date. This maintenance will entail an enormous effort.

Our current guess based on current growth patterns and computer sales is is that the net will increase to between 15,000 and 50,000 sites over the next five years. One reason is the increasing popularity of personal computers, whose owners will want the convenience and relative low cost of UUCP as an option. We must plan now for this explosion.

Once a map exists in machine-readable form, it is a simple matter to use that map to route mail. This means that instead of typing in long routes such as:

```
cbosgd!mhuxl!eagle!harpo!seismo!hao!hplabs!hpda!fortune!amd70!decwrl!joe
```

we type:

```
joe@decwrl.uucp
```

This not only saves a lot of typing but allows the machine to pick a "best" route. In this case it might have chosen decvax!decwrl!joe, a route the person may not have known about.

3. Becoming a Domain

For the past 12 years, the ARPANET has used a user@host mailing address syntax. A year ago they realized that a flat addressing structure such as this caused impractical maintenance problems; they then converted to a hierarchical user@domain syntax. The domain is a list of words separated by periods (e.g., F.ISI.ARPA) forming a hierarchy with the top level domain, ARPA, at the right. This method allows a very large number of hosts to be named without any host needing a complete list of all other hosts. Other top level domains, besides ARPA, are possible.

We propose to make UUCP a top level domain. This will enable users on UUCP machines to exchange electronic mail with users of machines in other ARPA domains, such as the ARPANET and CSNET.

In RFC 881*, the ARPANET sets forth a list of requirements for top level domains. These requirements are:

1. There must be a responsible person that can act as coordinator and answer questions. In addition to serving as a central contact point, this person will have the authority to enforce network policies and rules.
2. There must be a robust name service. There will be two separate name server machines to which one can send a mail message and receive information back about a particular site.
3. The domain must be of a minimum size. Since UUCP currently has over 2100 hosts, this should be no problem.
4. The domain must be registered with the central domain administrator.

We intend to meet these requirements. In addition, since we will be keeping a list of all host names, it will be possible to ensure that there are no duplications. The UUCP software allows host names of any length, but only uses the first six (System V) or seven (others) characters of the name. We will ensure that all host names are unique in the first six characters and are chosen from a set that will not cause trouble on other machines (lower case letters, digits, and a few punctuation characters such as hyphen and underscore).

The authority issue is slightly sensitive, since there is currently no authority over the UUCP network. Such ultimate authority is, however, necessary to protect other domains and other UUCP machines against unacceptable behavior. Such behavior might involve flooding another machine with large quantities of unwanted mail or generation of traffic that fails to conform to accepted standards and breaks programs on other machines. If a host continues to cause problems after repeated warnings, the site would lose their entry in the data base and their claim to their UUCP host name. We expect use of this authority to be extremely rare. The corresponding authority has existed on the ARPANET for years, yet no site has ever been disconnected because of it.

4. Pathalias

To solve the problem of determining "good" routes to foreign machines, Steven Bellovin has written a program called "pathalias" which he has placed in the public domain. Given a list of sites and their direct neighbors, pathalias computes an "optimal" route from the local host to each other host on the network. Upon receiving a new database, the site administrator runs pathalias on the database and stores the result in a routing file. Any program can then look up the best route to any site in the routing file.

Here is a sample database for a four host network. For each host, the name of the host is given, followed by the names of all sites to which mail can be sent directly. For example, allegra can send mail directly to all three of the other hosts, but gummo can only send mail directly to allegra. In this example, all links are two-way; in the real world, most but not all links are two-way.

```
allegra cbosgd, gummo, ucbvax
cbosgd allegra, ucbvax
gummo allegra
ucbvax allegra, cbosgd
```

The output from pathalias, run on host cbosgd, is:

* Request for Comments 881, Network Information Center, ARPANET.

```
0       cbosgd      %s
4000    allegra     allegra!%s
4000    ucbvax      ucbvax!%s
8000    gummo       allegra!gummo!%s
```

The first column indicates the "cost" (this is calculated using defaults); the second column is the machine name; the third column is the mail route to the machine.

Many circumstances can cause routing through a given machine to be more desirable (or less so). Some connections may cost more than others since some links go over leased lines, local area networks, or expensive long distance lines. Some sites may not have an autodialer or may not be able to afford long distance phone calls. Such sites cannot call their neighbors on demand, but must wait to be polled. The frequency of polling varies from every hour to seldom or never, depending on the amount of traffic coming from the other site. Some links are more reliable than others.

Pathalias can find the "lowest cost" path from the local site to each other host on the net if "costs" for connections ("edges") are supplied. Rather than use the default cost of 4000, as above, we can attach specific costs to each link. In the example below, DEMAND is 300, HOURLY is 500, DAILY/2 is 2500, and DAILY is 5000.

```
allegra cbosgd(DEMAND), gummo(DAILY/2), ucbvax(HOURLY)
cbosgd  allegra(DEMAND), ucbvax(DEMAND)
gummo   allegra(DAILY/2)
ucbvax  allegra(HOURLY), cbosgd(DAILY)
```

The pathalias computations become:

```
0       cbosgd      %s
300     allegra     allegra!%s
300     ucbvax      ucbvax!%s
2800    gummo       allegra!gummo!%s
```

Changing the "local host" to ucbvax yields:

```
0       ucbvax      %s
500     allegra     allegra!%s
800     cbosgd      allegra!cbosgd!%s
3000    gummo       allegra!gummo!%s
```

The pathalias program can also compute paths to other networks. We believe it is a powerful enough tool to solve the routing problems -- once an accurate database is collected.

5. Data Collection

Currently, there are a few databases scattered throughout the network. Karen Summers-Horton keeps track of those Usenet sites which receive the newsgroup net.announce. Their information includes the site's name, the contact person, electronic and US Mail addresses, and hosts with which the machine exchanges news. Several sites keep track of extensive L.sys files and can call hundreds of machines. These sites have an easy time of finding routes. Rob Kolstad maintains a list of network "edges": connections between pairs of machines. The list is not of ultimate utility since it contains neither connection frequency nor direction. Even though some hosts call others on demand, it is not a good assumption that mail can flow back the other way.

Rob Kolstad (allegra!parsec!kolstad, CONVEX Computer Corp., Dallas), Scott Bradner (allegra!wjh12!sob, Harvard University), and possibly a handful of other volunteers are currently involved in an effort to collect an accurate enough database to generate pathalias style routes for any given machine. Here is a typical entry in their database:

```
=Name:          parsec
=Machine Type/OS: VAX780/4.1cBSD
=Organization:    PARSEC/Convex Computer Corporation
=Contact Person:  Rob Kolstad
=Electronic-Addr: parsec!kolstad
=Phone:          214-669-3700
=Postal-Address:  1819 Firman #151, Richardson, TX  75081
=Long/Lat Coors:
=Comments:
=Editor:         parsec!kolstad Tue Jan  2 09:07:00 1984
=
===============================================================
#
parsec  unmvax(DAILY/3), rice(DAILY/3), uiucdcs(DAILY/3), ctvax(DEMAND),
        rice(DAILY/3), allegra(DAILY/3), dj3b1(DEMAND), smu(DEMAND)
```

The entries are stored in compressed format (but easily expanded). They have enough information for pathalias to make optimal routes and also include enough site information to solve most problems that come up that require contacting the site.

The database currently contains entries for over 2000 sites. Typically they look like this:

```
=Name:          acsa
=Machine Type/OS:
=Organization:
=Contact Person:
=Electronic-Addr: acsa!root
=Phone:
=Postal-Address:
=Long/Lat Coors:
=Comments:
=Editor:
=
===============================================================
#
acsa    inuxc(DAILY)
```

It is our goal to fill in each of the templates. We currently intend to do this by electronically mailing out our (sometimes partially completed) templates to each machine with directions for completing the form and returning it. We have several scripts and programs to maintain the database and automatically send out mail. We believe the initial data gathering effort will require from 3-9 months.

6. Data Base Maintenance

6.1 Short Term

Short term maintenance of the database will be done strictly by hand. Sites will be encouraged to mail corrections to a central collection point, at which time we will manually update the database. As this will take a considerable amount of time and effort, steps are being taken to automate the process as soon as possible.

6.2 Long Term

In the long term, it is hoped that we can distribute programs (or shell scripts) which will allow simple updating to the database by mailing well-formatted updates to an alias. While this scheme might still require human intervention in the form of "approving" updates, and dealing with improperly formatted requests, it still removes much of the effort from the update process.

We must also develop a "registry" for new sites to consult to verify uniqueness of their site name and enter their initial routing data. While this removes some of the anarchy from the network, it is felt that it is nevertheless a valuable step.

7. Data Base Distribution

Our current database is approximately eight megabytes and growing rapidly, so it is much too large to post to Usenet (even once). However, it is crucial to distribute enough data on a regular basis to allow mail routing programs to work.

One very simple option is to include the database on the USENIX tapes. This option has problems related to timeliness, but an out-of-date database is far more useful than none at all, and it would allow distribution of the entire database.

Another option is to post (on a regular basis) complete information for a small set of backbone sites to Usenet. In addition to this complete information, we would also post a list of all other hosts, and a path from each host to a backbone site. This should cut the size of our initial distributions by a large factor.

A third option is not to post the entire list of host names. Instead, a hierarchical structure would be worked out in the spirit of ARPANET domains. Thus, an address like cbosgd.uucp might become d.osg.cb.att.uucp, indicating that within the UUCP domain, AT&T region, Columbus location, OSG project, the D machine is specified. No fixed structure such as id.proj.loc.reg.UUCP is implied, each subdomain would subdivide itself as appropriate. The Postal Service and Telephone Company have set up similar hierarchies that can route traffic without any complete lists of all possible locations.

The last two schemes have the disadvantage that they do not explicitly take advantage of the richness of connections that do exist. These problems remain to be solved.

Finally, the name servers to be set up would provide for 'on demand', electronic distribution of directory information for individual sites.

8. Summary

As it stands, the UUCP network is a growing, viable entity which provides very low cost network facilities to well over 2,000 machines. The problems of unreliability, lost messages, and variable propagation delay have been tolerated for many years. Because of the network's growth, its machine-to-machine routing scheme, and lack of centralized control, the requirement for an accurate network-wide map is essential. We need the cooperation of each site in order to construct and maintain a reliable database.

Building Tunnels and Bridges: Constructing a Commercial Application Under UNIX

Ellen Ullman, Page Thompson, Jerry Carlin
Insurnet, Inc.
1900 Powell Street
Emeryville, CA 94608

BUILDING TUNNELS AND BRIDGES: Constructing A Commercial Application Under Unix*

Ellen Ullman
Page Thompson
Jerry Carlin
Insurnet, Inc.
Emeryville, California

ABSTRACT

This paper discusses the necessity of building a high level programming environment before a commercial application system can be developed efficiently using Unix/C. Both the process leading to the construction of the applications environment, and the environment itself, are described.

> "The dearth of mature Unix-based commercial business applications is considered one of the prime factors inhibiting the growth of Unix."
>
> Michael Azzara,
> Computer Systems News, 1983[1]

> "A great deal depends ... on the Unix software houses: if the visible output of Unix-based software doesn't grow, word may spread that there's no software available for Unix."
>
> David Fiedler, BYTE, 1983[2]

> "Those of us not involved in document preparation tend to use the system for program development, especially language work. There are few important 'applications' programs."
>
> Dennis M. Ritchie and Ken Thompson,
> Communications of the ACM, 1974[3]

----

* Unix is a trademark of Bell Laboratories

1. INTRODUCTION

Nine years after Dennis Ritchie and Ken Thompson described Unix as having "few important applications programs", trade publications still refer to Unix as a system in search of applications software. The release of new software offerings has not kept pace with new implementations of Unix. As members of a team building a major commercial application under Unix/C, the authors of this paper are in a unique position to understand the reasons for this "dearth of mature Unix-based commercial applications".

In March of 1981, Insurnet, Inc., a turnkey vendor of software to the insurance industry, began to consider duplicating all of its existing software from a Pick* to a Unix environment, and to develop new and enhanced software under Unix. A plan for implementation of Insurnet software under Unix was produced in May of 1982.[4] Under the implementation plan, the system was to be released for beta test in January of 1984. Now, the Unix development team has just completed a representative prototype system and expects to release the beta system no earlier than the end of 1984. While delays of this type are not uncommon in software development, a large part of our difficulties stemmed from the current state of the Unix programming environment as it applies to the production of integrated, data-driven systems geared for the commercial end-user. While we are now certain we will be able to release a powerful, state-of-the-art system, at times we believed we had walked head-on into every reason behind the lack of commercial business applications under Unix.

Before we could proceed to develop our application, we first had to develop an environment congenial to commercial data processing. We did not plan to spend a year constructing an environment in which applications programmers could function; it was forced upon us by the incompleteness and lack of integration of the tool set available to us. As applications developers in a commercial environment, the pressure was upon us to produce a marketable system. It was with great difficulty that we presented reports, month after month, stating progress on "screen I/O methodologies" and "database integration", rather than on policy management and accounting systems. However, we had to resist the pressure

----

* As used here, "Pick" refers to a group of operating systems based upon the system originally developed by Dick Pick for Microdata Reality, now running on Honeywell Level 6 (Ultimate), IBM Series I, Altos 586 and some 68000-based systems.

to just start writing code, in order to build, or acquire, or integrate the tools that would enable us to do just that.

We have created a system development environment in which the hypothetically "average" commercial programmer can function with a high degree of productivity. Ironically, the creation of what amounted to a higher level language, shielding the programmer from the complexities of Unix/C without sacrificing programming power or efficiency, was possible only because of Unix's malleable nature. We were able to use Unix/C to create an environment that appears to the programmer as "not Unix/C". Why we should want to create a "not Unix/C" environment will be clear if it is remembered that the average commercial programmer is accustomed to working with very high level I/O and intrinsic functions, and is often unacquainted with such C language features as address pointers, registers and bit manipulation. We had two choices: hope for the existence of a pool of highly trained C programmers who would wish to work on insurance applications, or create an environment in which programmers already involved in insurance applications could work effectively. For obvious reasons, we chose the latter course.

The system development environment we have created is tailored to our own requirements as a turnkey vendor; it may not be applicable for every Unix applications shop. We believe that what is applicable to others is the process through which we realized the need for our own applications environment and then, finally, proceeded to create it. Our experience should illuminate the status of Unix as an environment for commercial applications and should benefit other Unix applications developers. In addition, we hope our experience will present a thorough set of requirements to developers of database management systems and programming productivity tools, making the way a little easier for the Unix applications developers who will follow us.

2. DESCRIPTION OF THE APPLICATION SYSTEMS

Insurnet, Inc. develops software for use by independent insurance agents and brokers. Insurnet software automates agent and broker offices and provides data communication interfaces between the agent/broker office and insurance companies. Major application modules include policy management, claims reporting, accounting, marketing, word processing and end-user personal database capabilities. Also included are batch (3780) and interactive (3270) interfaces with major insurance carriers. Except for the stand-alone word processing and personal database modules, the subsystems are integrated, working from a single database.

The Insurnet system that currently runs in the Pick environment has over 1000 programs, 150 data entry screens, and 120 record types with over 3500 data elements. Typical users have 10 work stations and may have databases of up to 100 megabytes. (Storage requirements are for the Pick system which uses variable length fields and records; database size could increase dramatically under a fixed length field and record format.) The system spends much of its time managing the entry and retrieval of data; I/O and database organization are the cornerstones of the system design. Due to the complex of laws and regulations that surround both insurance and accounting, precise field and relational edits and database integrity are of paramount importance.

The existing system design relies heavily upon built-in database capabilities offered by the Pick system: an ad hoc query language integrated with a data dictionary, the ability to form run time indices on any data element in a record, and variable length fields and records which permit large record formats without inefficient use of storage. In addition, the programmers who maintain the existing system, and who would have to form the pool of talent available for migration of the system to Unix, have been shielded from many of the physical realities of the computer upon which they are working: Pick's DATA/BASIC language is not a typed language; the programmer need not declare a variable's type nor understand how it is handled internally.

3. THE DECISION TO USE UNIX

Given the nature of the application system we were to convert--heavily dependent upon I/O and database capabilities, written in a language without data types--it can be understood that the decision to use Unix and C was not entirely based on technical considerations. Were it not for overriding business factors, Unix/C--without database capabilities, with only low level I/O, strongly typed--might not have been the environment of choice.*

The business factor that ultimately led to the choice for Unix was, naturally, profitability. Insurnet was formed

----

* It is important to note that Insurnet's operating systems people, as opposed to applications designers, were highly in favor of Unix. The distinction here is between an evaluation of Unix as an interesting operating system and as an environment permitting rapid applications development.

three years ago as a joint venture of American Information Development (the predecessor company, formed in the early 197Øs), Quotron Systems, Inc. of Los Angeles and the Continental Corporation. Using Unix, Insurnet would be able to exploit the economic advantages of utilizing hardware, operating system software, field support, and communications capabilities provided by one of its partner companies, Quotron.

At the inception of the Insurnet joint venture, a project was started to migrate American Information Development's software to Quotron's hardware, a new machine with a 68ØØØ processor then under design by Quotron. The next decision was whether Insurnet would use Quotron's choice for an operating system as well. Quotron chose Unix System III as a base for its applications. Since American Information Development's 1ØØØ programs and large installed user base represented a substantial investment in the Pick operating system, the decision to migrate to Unix was not made lightly.

In an effort to preserve existing code, Insurnet first investigated financing the porting of the Pick operating system to the 68ØØØ processor, or a Pick look-alike system to be written "on top" of Unix. Arguments against these approaches, besides the high cost involved, were potential inefficiency of piggy-back operating systems and the timing of new operating system development. In addition, it seemed more prudent to rely upon the expertise of a partner company, Quotron, rather than rely upon third parties. What finally solidified the decision for Unix, however, was the desire to improve the system's reliability and maintainability. As stated by Carl Poulsen, Insurnet Senior Vice President and designer of the original software, "Insurnet's software was designed seven years ago and continues to satisfy the current marketplace more because of ingenuity and innovation than because of continuing soundness of initial design."[5]

Having decided to cast its fate with Unix, Insurnet was not blind to the difficulties that the Unix environment would present. The "Insurnet Advanced Product Line Project Summary" that launched the project noted that "programming in C is more difficult than programming in DATA/BASIC", but assumed that those difficulties could be minimized due to C's ability to look like other languages if so desired; we would make C look like DATA/BASIC. We did not realize the extent of the transformation that would be necessary.

4. GROWING OUR OWN UNIX EXPERTS

> "Programming environments usually reflect the size, organization, and sociology of the groups that create them."
>
> Brian W. Kernighan,
> John R. Maskey[6]

Implicit in the plan to make C look like DATA/BASIC was the desire to retrain Insurnet's Pick programmers and turn them into C programmers. The rationale behind this was: a large pool of skilled C programmers does not exist; what skilled C programmers do exist come from systems backgrounds, where they have been writing things like device drivers and compilers and kernel modifications, or from academic environments, where they have had no exposure to commercial data processing systems. What were called Unix/C "applications" were in fact generalized tools like word processcrs, database management systems and spreadsheets. As much as we would have liked to be rescued by hiring expertise, we were left staring at a contradiction: the lack of existing Unix/C commercial applications also meant the lack of existing Unix/C commercial applications programmers. The talent we needed simply didn't exist.

The original project team consisted of a senior vice president, a director of applications development, a director of operating systems maintenance, and a secretary. They all started by writing the first exercise given by Kernighan and Ritchie in The C Programming Language: the little program that prints "hello, world".[7]

From that point, it was a matter of designing a training curriculum and picking the programmers who were to staff the project. The programmers were chosen using the following criteria: First, each had to want to learn Unix and C; those programmers who felt committed to Pick were not pressured to transfer to the project. Second, thorough knowledge of the existing application system was required. Above all, the programmers chosen had to be confident of their abilities in order to weather the difficult process of retraining, of suddenly losing expert status, and of building a new application system in spite of expected frustrations.

The expected frustrations arrived. Senior programmers, who considered themselves highly skilled, suddenly did not know how to write a program to input a single field from a terminal and store it on a disk. Programs failed and could not be debugged, due to a lack of knowledge about internal addressing schemes. Seemingly inexplicable problems were tracked down, after days of frustration, to a missing null terminator or blown array subscript. The compiler was not protecting them.*

A programmer using a high level language like COBOL or BASIC can study the manuals and then go off and write code, confident that the manuals contain the full range of available syntax. Using C, the final syntax is created piece by piece, through the writing of functions. This means that any one individual programmer is highly dependent upon the output of other programmers, each of whom is writing functions that will be used by the others.

This loss of self-sufficiency is a formidable hurdle for programmers experienced in languages with a fixed syntax. One's bugs may in fact be caused by the bugs another has left in a function; then again, they may not. The high degree of friendly collaboration and tolerance this requires was not always observed. Two team members quit and had to be replaced; others threatened resignation; tempers flared; certain individuals refused to work with certain other individuals.

Indispensable to the success of the project was the formation of a cohesive team from ten highly motivated and dedicated individuals. This could not have been done without the understanding at the highest level of management that immediate productivity is not the hallmark of success in a lengthy, groundbreaking project. Unless senior management clearly recognizes that the team must be insulated from normal business pressures for some period of time, a project of this nature should not be attempted.

5. EARLY FAILURES

The initial ignorance of team members in relation to the task ahead is best characterized by a single Insurnet document, C Programming Guidelines, released in August, 1982. A memo accompanying the distribution of the Guidelines states,

----

* "Much of the C model relies on the programmer always being right ... the amount of freedom provided in the language means that you can make truly spectacular errors, far exceeding the relatively trivial difficulties you encounter misusing, say, BASIC." Stephen C. Johnson and Brian W. Kernighan, "The C Language and Models for Systems Programming", BYTE, August, 1983, pp. 48f

"The ... manual really defines an application language which will be valuable to any business which wants to write in C as productively as they currently write in BASIC or COBOL."[8]

Although this statement shows that we correctly understood our goal, we we were much further from its accomplishment than we knew. We had yet to realize that a standardized subset of C, coupled with the careful use of existing Unix tools, would not come close to duplicating a business applications environment.

In an attempt to match the language to our programmers, we began by forbidding the use of: unary plus/minus operators, bit shifts, the bitwise and/or, pointers, register variables, functions that returned values and embedded assignments.* In other words, we were attempting to avoid programming in C. The folly of this approach became clear as members of the team began to become more proficient in C and began to sneak "forbidden" code into their programs. Without giving up our goal of creating a higher level environment, we had to admit that simply trying to ignore a good portion of the C language was fruitless.

The initial intent of the C standardization effort was to create a semantic masquerade: if, through the use of restrictions, redefinitions and functions, C could be made to look like BASIC or COBOL, then commercial programmers could be transferred to the new environment with minimal retraining.

The extreme to which this concept was taken can be illustrated by the attempt to duplicate the DATA/BASIC "GOSUB" statement. A good deal of time was spent debating how best to express GOSUB{numeric label} in C. This attempt to preserve an anachronistic programming statement should clarify our desire to retain a familiar environment.

A second fundamental misconception was of the "cart before the horse" variety. If C could be manipulated into imitating a higher level language, then a supporting environment could be created using Unix tools, primarily SCCS, make and lint. Months after this concept was endorsed, we saw the inherent error: what we had considered to be an applications development methodology was in fact the environment surrounding the finished programs; what we really needed was something to help us write the programs in the first place.

----

* The statement: while ((c = getchar()) != EOF) is an example of the forbidden embedded assignment.

6. SELECTION OF A DBMS

It was accepted from the very inception of the project that the database capabilities built into the Pick system would have to be imported into the Unix environment. Thinking it unlikely that any single DBMS would meet Insurnet requirements, we initially planned to write our own. However, the amount of time required to do so and our lack of in-house expertise, soon brought us to the decision to purchase a DBMS and source license.

We examined Ingres, Unify, Informix, Logix, Oracle and Sequitur and eventually chose Unify, mostly because of performance considerations. Our selection of Unify is not a blanket endorsement. As will be seen below, Unify turned out to be unsuitable to some degree. However, given that we provide turnkey software, Unify was the best match for our requirements.

As a provider of turnkey systems, we were looking for a DBMS that was not geared solely, for the end-user, but that provided a substantial C language interface. Unify was actually built by a turnkey software vendor; a library of C language functions came with the system. Since we would be building an application on top of the DBMS, performance was of primary importance to us. We were not looking for run time flexibility at the expense of performance. That Unify required a system reconfiguration after each database change was no of concern to us; the database schema is stabilized before installation at our users' sites.

7. WE HAVE MAKE, WE HAVE SCCS, WE HAVE CURSES, WE HAVE TWO FORMS PACKAGES, WE EVEN HAVE A DBMS: WE STILL DON'T HAVE A SYSTEM

> "Almost everything you could ever need to develop and use a software system on Unix is on the market; you just have to find it."[9]

It would seem, once we had purchased a DBMS, that all the pieces were in place for the construction of a system. What else could we possibly need in the way of development assistance? The answer was: integration of the available tools.

First, Unify screen I/O operations were found to be inadequate. Cursor motion is not significantly optimized. At 1200 baud, one can see Unify screen I/O functions printing blanks over blanks, slowing down terminal operations. Since many Insurnet users do not have in-house computers, but communicate over 1200 baud lines to regional processors, Unify screen routines were considered unacceptable. In addition, Unify screen I/O functions provide no facilities for multiple window operations or visual attributes. The problem here is not that the Unify designers did not think of everything; the problem is that they assumed their functionality was complete and provided no way to "plug in" other modules at low levels. The lack of designed-in hooks, the inability to make one tool interact with another, is a common feature of most system development tools.

Two forms packages were available to us, one from Johns Hopkins[10], and one under development at Quotron; both lacked the means to work with outside modules. The Quotron package, having sophisticated terminal handling abilities, had no DBMS interface. The Johns Hopkins package had neither multiple window nor database integration capabilities. The inability of the form to know about the existence of the database was considered unacceptable; it negated the purpose of the database design.

We were determined to have the features offered by the "curses" screen handling package: physical terminal independence (terminal capabilities library, "termcap"), cursor motion optimization, and multiple window operations.[11] Physical terminal independence was a foregone conclusion; as a turnkey vendor, we could not predict the hardware configuration that would be used over the life of the system. We would have liked the visual attribute capabilities of a Quotron screen handling package then under development. Since we were building an application with over 150 data entry screens (growing yearly as the scope of insurance automation proceeds), we absolutely required a facility for defining screen forms external to the C program.

The tools were all there, but they would not work with each other. In fact, Unify screen I/O functions could not even be loaded in the same module as curses functions, due to inconsistencies in the use of the "termcap" library. Deciding for cursor optimization, we completely abandoned Unify screen I/O routines. In giving up Unify screen I/O, we were losing a substantial portion of the C language interface that was to present high level syntax for our programmers. We were back to the problem of not being able to input a single field from the terminal and store it on the disk.

In addition, Unify was not configured to handle really large databases, such as ours. While we were able, without great difficulty, to modify the system to accept the entry of a large data dictionary, once our data dictionary grew to a certain size, we could not compile any programs that used the database. Unify field and record names are "#defined" and then handled as integers internally. In order to use the record and field names in applications programs, the program must "#include" a header file containing the "#define" statements. We quickly reached the limit of "#define" statements; the preprocessor promptly quit, saying, "too many defines".

At that point, we were past wondering if we could give up the desire for some nice features in order to accommodate ourselves to our tool set. We knew we had no choice but to begin to build an environment more to our own specifications. However, we also knew we did not have two years to rewrite the DBMS. The dilemma was to modify the environment without having to rewrite the heart of any single tool. When we described our problems to our counterparts at Quotron Systems and the solutions we were using, they described our approach as "building tunnels and bridges". That is exactly what we have been doing: going around our tools where we can't use them, connecting one tool to another where we want to use them together.

8. THE APPLICATIONS DEVELOPMENT ENVIRONMENT

8.1 The Data Dictionary Extension

The Unify data dictionary defines a data field by its data type, its storage length, and its dependency upon a parent field, if any. While basic type and length edits are essential, we determined that the Unify field definitions did not go far enough towards providing a standard field edit set. We decided to create an extension to the Unify data dictionary. For each field in the database, we store a data definition record; we consider the maintenance of the data definition record to be part of database schema maintenance.

The Insurnet data definition record stores three essential pieces of information about the data field:

1. Insurnet Edit Type - Our standard header file currently defines the 14 edit types listed below. This list has changed over the course of our system design, and may continue to do so as requirements change or are clarified. Each edit type (except for "no edit") corresponds to a single field edit function. The advantages of having a single edit function for all generically related fields are: the

removal of repetitive code from individual programs, uniformity of the user interface, ability to respond easily to desired changes in the handling of the edit.

```
#define EDNONE    0     /* no edit */
#define EDPHONE   1     /* phone number */
#define EDDATE    2     /* date */
#define EDMONEY   3     /* money */
#define EDFLOAT   4     /* floating point */
#define EDINT     5     /* integer */
#define EDTIME    6     /* time */
#define EDALPHA   7     /* alpha */
#define EDPATTERN 8     /* pattern match */
#define EDYN      9     /* y/Y or n/N */
#define EDDRCR   10     /* debit/credit, DR or CR */
#define EDTABLE  13     /* table lookup */
#define EDZIP    14     /* zip/postal code */
#define EDFILE   15     /* must be on file in */
                        /* another relation  */
```

2. Acceptable Field Tokens - Under our design methodolgy, an operator's entry may be considered acceptable in one of two ways: either data is entered that passes the appropriate field edit function, as above, or the operator enters an acceptable field token that indicates some action other than immediate data entry is desired. As was true with edit types, the list of field tokens has grown as applications development has proceeded. A partial list of currently used field tokens is given below:

   EDREFL - a "reference list" is desired. Certain fields, such as customer numbers, are considered "reference fields"; that is, the customer is identified in a customer relation and thereafter referred to by number. Often, operators are not certain of the correct number or code. If a reference token is applicable for the field, the operator may enter the token, search the appropriate set of reference data, select a code or number, and be returned to original program. The use of the "reference list" facility provides a user friendly method of handling coded data while preserving database normalization.

   EDSPECL - a "special list" is desired. The system provides a variety of "lister" facilities, for example, a list of all the automobiles or drivers on a single auto insurance policy. These "listers" operate in a manner similar to the reference listers above, but are handled using a different search methodology.

EDEXIT - the operator would like to exit the program at the point this token is entered. Since insurance applications often involve long data entry sessions, a facility was provided to permit program exit where such an exit would be non-destructive to database integrity.

EDNEW - the operator would like to enter a new reference item. If, for example, the program at hand is prompting for a customer number and the appropriate customer has not yet been established in the database, the operator may enter this token, be taken to the customer maintenance program (if security permits) to add the customer, and then return to the original program.

The field tokens are handled as bit flags, or'ed together where more than one token is acceptable. While certain of these tokens would appear to be applicable only on a screen-by-screen basis, research on our existing application system has indicated that these system-wide definitions can be made on a data element basis with 99% applicability. After much debate about the merits of letting programs override database schema definitions for screen-by-screen exceptions, we came to the conclusion that it was not a sin to let some intelligence reside in the programs.

3. Help Messages - The system recognizes a help token, which is always an acceptable entry. The help message is displayed on the system-defined "error row" when the help token is entered.

The Insurnet data dictionary extension is one of the "bridges" that connects the DBMS to our data handling functions. Ideally, the specification of the complete set of data-defining information should be done in one operation. Having an "official" data dictionary and a separately-maintained extension is less than efficient. DBMS developers might consider providing locally defined edit functions and/or data types as part of database specification process, thereby building in a bridge to a variety of applications.

8.2 The Data Dictionary and Internal Buffer Variables

The theoretical goal of a DBMS is to provide data independence between the data dictionary and the programs that use the database. If the DBMS host language is not one that is strongly typed, the management of program buffer variables is not a problem that must be addressed by the DBMS. However, with C as the host, a strong link between the data dictionary and internal variables is absolutely required.

The necessity for this strong link can be seen in the case of integers: under Unify, changing the input length of a numeric field from 5 to 9 changes its type from short to long. In practice, this means that all programs that use the data element would have to be modified.

Insurnet had an additional impetus to devise a link between the data dictionary and internal variables. Insurnet's Pick-trained programmers have been shielded from awareness of internal data types by the DATA/BASIC language.

In order to link the programs to the data dictionary, Insurnet has written software that creates internal buffer variables that are consistent with database definitions. A header file is created for each record type defined in the database. The header includes: 1) the #defines for the database field names in the record, thereby solving the "too many defines" problem for a large database (please see section 7), and 2) a structure for the record, with a member for each field in the record. A sample header is shown below.*

```
     #define AGTNO          57
     #define AGTNAM1        58
     #define AGTNAM2        59
     #define AGTSTR1        60
     #define AGTSTR2        61
     #define AGTCITY        62
     #define AGTST          63
     #define AGTZIP         64
     #define AGTDIV         65
     #define AGTSNAM        399
```

----

* In examining the structure, please note that the structure tag, "agt", as well as the declaration of the structure variable "agt" is not accidental. In many cases, data entry screens must handle multiples of a single record type, as in the entry of four drivers listed on an auto policy. Given the tag, the programmer can then declare additional structures as needed.

```
struct agt {
        short   no;
        char    nam1[32];
        char    nam2[32];
        char    str1[26];
        char    str2[26];
        char    city[22];
        char    st[4];
        char    zip[10];
        short   div;
        char    snam[16];
} agt;
```

Once the header is included in the C program, the programmer is presented with a set of internal variables that are of the correct data type and that are named in accordance with data dictionary naming conventions. For example, the database field AGTNO has a buffer variable that is named agt.no in every program that uses the AGT record, thereby providing automatic adherence to naming conventions.

8.3 The Screen Form Extension

As discussed above in (7), we required a screen formatting tool that would enable us to define screen layouts external to the application program, and that also interacted with the data dictionary. The screen form tool provided by Unify, SFORM, was satisfactory only to the extent that it satisfied our minimum needs: formats specified externally and linkage to the database.

SFORM stores a limited amount of information about each field on the screen: its input coordinates, its data type, the underlying database field, the screen prompt and the prompt's coordinates. The screen form information is loaded at run time and the Unify screen I/O functions then interact with the screen data. From Insurnet's point of view, this tool was unsatisfactory because: the Unify screen I/O library was not acceptable to us due to lack of effective cursor motion optimization; the loading of screen data at run time was considered to be inefficient; and, of course, SFORM could not possibly interact the the Insurnet data dictionary extension.

Another element in the Insurnet environment is a program that interconnects the screen designed under SFORM, the data dictionary extension and internal buffer variables. This program creates header files, one for each screen, and the header files are then included in C programs. In this way, the three objections listed above are overcome: the screen format is now available to any set of functions; screen data is compiled with the programs and loaded without extra disk reads at run time; the dictionary extension is part of the screen field's definition. At the risk of straining our metaphor, the program that creates the screen headers (makescrh) creates a "bridge" between SFORM and the data dictionary extension, between the Unify dictionary and internal variables, thereby "tunneling around" the Unify screen I/O library.

The program that creates the headers initializes a screen definition structure for each field on the screen. The format of the structure is:

```
     typedef struct{
          int   inx;          /* input x-coordinate              */
          int   iny;          /* input y-coordinate              */
          int   unifytype;    /* unify data type                 */
          int   edit;         /* Insurnet edit type              */
          int   length;       /* dictionary length               */
          int   blanklen;     /* blanking length; max display */
                              /* length; computed by makescrh */
          short flags;        /* field tokens                    */
          char  *buffer;      /* pointer to buffer variable      */
          int   dbfield;      /* unify field name (number)       */
          char  helpmsg[80];  /* help message                    */
          char  sprompt[41];  /* screen prompt                   */
          int   prx;          /* prompt x-coordinate             */
          int   pry;          /* prompt y-coordinate             */
     } SCREENDEF;
```

Please notice that there are two elements missing from this structure that might normally be considered part of the description of a screen: the screen heading and the visual attribute characteristics for the screen fields. Under Unify, the screen heading is a consequence of the menu selection list ("tunneled around" via a heading function, see below). In the case of visual attributes, we decided that the attribute to be used is more a consequence of the type of screen element than of any one particular screen field; headings, prompts, help messages, the current input line, each have their own type of visual attribute, to be handled by the functions that manage that classification of screen element.

As is the case with the data dictionary extension, we now have a two-staged operation for the specification of a screen: the entry into SFORM and the making of the screen header via Insurnet software. While we can smooth this situation with further software, a screen formatting tool geared for the developer of sophisticated applications is the long-range solution. The ideal screen formatting package would permit the user (that is, the application developer) to specify the structure that is to define a screen field.

8.4 The Function Library

The basis of the application environment is a function library that interacts with the headers described above. The goal of the function library is to provide the commercial applications programmer with the means to accomplish the majority of business programming tasks with a syntax that is both simple and descriptive.

For example, the main loop of a the typical data entry program illustrated below is brief and readable through the use of appropriate library functions. The screen template is displayed by the call to "prompts"; the template is reinitialized by the call to "clrdata". The work of editing field-by-field input and of updating the database is performed by the use of internal functions in a standardized fashion. Please note the "includes" of header files discussed above in sections 8.2 and 8.3.

```
#include "istd.h"            /*  standard header file     */
#include "infile.h"          /*  defines for record names */
#include "files/inmad.h"     /*  buffer variables         */
#include "screens/inmkmad.h" /*  screen header            */

....

main()
{
    bool getdata();
    bool updok;

    setup();
    head(stdscr, "mkmad", "Maintain Advertising");
    prompts();
    do
    {
        init(MADno, &madno);
        updok = getdata();
        if (updok)
        {
            update();
            clrdata();
        }
    } while (updok);

    eoj();
}
```

The code for the "getdata" function is shown on the next page. The calls to "infield" are particularly significant. Since the Insurnet application is one that is heavily centered upon the entry and editing of data, the foundation of the library is this function that accomplishes data entry validation. The "infield" function highlights the current entry line, manages the entry of field tokens and screen cursor motion tokens, edits entered data for type, length and Insurnet edit type, and converts the validated entry into internal format. A diagram illustrating the working of "infield" is given following the "getdata" code sample.

The "nextprompt" function controls internal loop iteration. This is done via manipulation of the variable "promptno" in accordance with rules based upon the next field in the screen definition array and the next required field.

Where exceptional field edits or relational edits are not required, library functions enable the programmer to handle data entry for a given field by a simple call to infield within the appropriate case label.

```
bool getdata()
{
    ....

    while (promptno <= SCREENCNT)
    {
        switch (promptno)
        {
        case MADno:
            ctl = infield(MADno, &madno);
            if (ctl == NLENT || ctl == BACKUP || ctl == LASTREQ)
                return(FALSE);
            read_ctl = keyacc(MAD, &madno);
            if (read_ctl == Ø)              /*  change mode  */
            {
                disfield(MADtype);
                ....
            }
            else                            /*  add mode     */
            {
                ....
                clrdata();
                outfield(MADno, &mad.no);

            }
            break;

        case MADtype:
            ctl = infield(MADtype, &mad.type);
            break;

        case MADnam:
            ctl = infield(MADnam, mad.nam);
            break;

        ....
        }
        nextprompt(skip, backup, p_order, ctl, &promptno);
    }
    return(TRUE);
}
```

INFIELD

Infield's syntax is:

```
     infield(screenfield name, buffer)
     int screenfield name;
     char * buffer;
```

The screenfield name is "#defined" by SFORM and serves as the index into an array of SCREENDEF's. The above syntax is converted into:

```
     winfield(stdscr, &SCREENDEF[screenfield name], buffer)
```

via a macro. As with "curses" functions, infield operates on the default standard screen (stdscr) or on the named window by use of the "w" function, as in: winfield(window, screenfield, buffer).

Infield uses "curses" functions to get a string through the named window or stdscr. Infield checks for the entry of screen or field tokens, does a length check, a basic data type check, calls the appropriate Insurnet edit function and converts the entry into internal format, placing it in the buffer provided.

The table below is a high-level summary of the work performed by the main loop of infield. This function and the associated conversion and edit functions together comprise over 1500 lines of source code.

TEST	ACTION	ADDITIONAL COMMENTS
Error message on screen	Clear message	
Screen control token input	If token allowed return token, else display error message, restart loop	"Tokens" denote terminal-independent screen manipulation characters, such as: backup to last field, go to next required field, exit, etc.
Help requested	Display help message, restart loop	Field-specific help messages are contained in the SCREENDEF structure.
Data input fails length oheck	Display error message, restart loop	
Caller in change mode, NULL entered	Set 'no change' flag	Adding new data and changing existing data are handled by the same module.
Delete requested and field optional	Null the field	
Field type = DATE and field previously empty and NULL entered	Generate today's date	This is an example of detail-level processing performed by the function.
Field is required and NULL entered	Display error message, restart loop	
Data input	Call appropriate edit function	The edit type of the field determines the function to call (ie., date, int, money, float, table, file lookup,etc.) Field-specific edit types are stored in the SCREENDEF structure.

In addition to managing data entry and retrieval capabilities, these "high level" functions interact, in turn, with other function libraries, such as that of Unify and "curses". The function library thereby shields the programmer from the specifics of disparate, lower level libraries. The high level functions integrate the individual libraries that are in use, coordinating their conventions and syntax. Return values are standardized; syntax is simplified where possible by the use of macros. The patchwork nature of the environment, the fact that it rests on the use of multiple unrelated tools, is invisible at this level.

8.5 A High Level Applications Language

The result of our efforts has been the creation of a high level language suitable for use by commercial programmers in the construction of complex business applications. Using this language, we have successfully produced a number of prototype sub-systems and are currently in the process of coding production applications.

The fact that it is possible for us to produce such applications code, in a timely fashion, with a staff that has been self-trained in Unix/C, is the measure of our success. We have built an environment wherein an applications programmer can progress from database schema design through coding of the source program in an orderly and standardized fashion. From a high level view, the steps involved are:

- enter/change database schema
  enter/change dictionary extension
  reconfigure database

- create program buffer headers

- enter/change screen layout
  create screen header

- construction of source file,
  using program template and function library

Our experience over the last year and a half has enabled us to define the following capabilities as benchmark requirements for a high level environment:

- data dictionary capabilities that permit the applications developer to specify additions to the set of data-defining information.

- sophisticated screen I/O capabilities, including cursor motion optimization, multiple windows, terminal independence, and video attributes; ability for interaction with the data dictionary of choice; ability to specify screen field-defining information.

- database storage and access capabilities which do not strain under the weight of large databases and complex data relationships.
- a function library which may be expanded.
- integration of capabilities such that the application programmer sees only a unified, appropriate tool set.

9. CONCLUSIONS

While Unix offers the systems programmer a powerful and elegant environment, a practical environment for the development of commercial applications is not yet generally available. Pipes, filters and shell scripts are not substitutes for task-specific code; programs still have to be written. The programmers who will write those programs will come from the ranks of today's COBOL and BASIC programmers and the level of the Unix environment will have to be made "higher" to accommodate them.

At the current state of the art, a company undertaking the development of a complex business application under Unix must be prepared for a long start-up period while personnel train themselves and create the tools necessary for applications development. Such a development effort will be costly, and the development environment produced will be, necessarily, less than perfect. This will be the case until sophisticated applications development tools, geared for the commercial programmer, have been produced.

Unix is a leading candidate for the operating system of choice in the fast-emerging world of multi-user super micros. Whether or not Unix can now move out of the academy, where it gained its initial popularity, and into the general commercial world is the question. Our efforts are proof that it is possible today to create commercial applications under Unix. However, the success of Unix as a general use operating system depends, ultimately, upon its acceptance by the business data processing community.

REFERENCES

[1] Azzara, Michael, "AT&T Launches Bid to Buy Unix Software", Computer Systems News, October 31, 1983.

[2] Fiedler, David, "The Unix Tutorial Part 2: Unix in the Microcomputer Marketplace", BYTE, October, 1983.

[3] Ritchie, Dennis M. and Thompson, Ken "The Unix Timesharing System", Communications of the ACM, July, 1974.

[4] Poulsen, Carl, "Insurnet Advanced Product Line Project Strategy", Insurnet, Inc. internal paper, May 24, 1982.

[5] Poulsen, Carl, ibid.

[6] Kernighan, Brian W. and Maskey, John R., "Software Development Environments", IEEE Computer Society Press, pp. 152f, 1981.

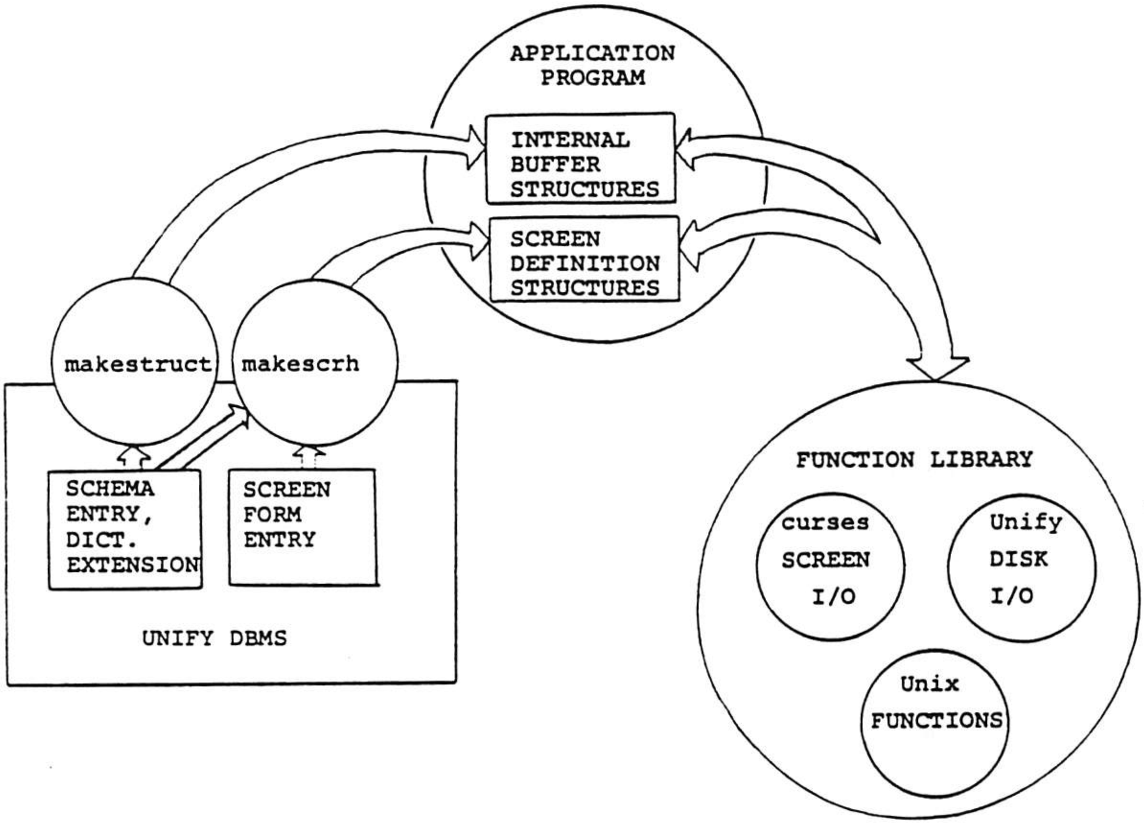
[7] Kernighan, Brian W. and Ritchie, Dennis M., The C Programming Language, Prentice Hall Inc., 1978.

[8] Insurnet Advanced Product Line Memo #13, Insurnet Inc. internal memo, July 28, 1982.

[9] Fiedler, David, "The Unix Tutorial Part 1: An Introduction to Features and Facilities", BYTE, pp. 186f, August, 1983.

[10] Diaz, Mathew, "The Forms Package Version 2.8: Programmers Guide", The Johns Hopkins University Applied Physics Laboratory, March 4, 1982.

[11] Arnold, Kenneth C.R.C., "Screen Updating and Cursor Movement Optimization: A Library Package", Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California.



A HIGH-LEVEL APPLICATIONS LANGUAGE

This diagram depicts the interaction of the various components used by Insurnet to construct an integrated programming environment.

Life with UNIX in Real-Time

Steven T. Polyak, Jeffrey S. Barr
Contel Information Systems, Inc.
Software and Systems Division
4330 East-West Highway
Bethesda, MD 20814

LIFE WITH UNIX
IN REAL-TIME

Steven T. Polyak
Jeffrey S. Barr

January 20, 1984

Software and Systems Division
Contel Information Systems, Inc.
4330 East-West Highway
Bethesda, MD 20814

INTRODUCTION

Much has been said and written about what can and cannot be done with UNIX in a real-time applications environment. Here we present a case study, one that addresses most of the important topics that arise when one wishes to adapt UNIX to such a real-time application. Specifically, we will address inter-process communication, a commonly addressable data base in RAM, priority scheduling problems, and the use of the UNIX kernel for special non-interruptable operations.

Time-Frame: Early 1982

Problem : We needed a Multibus-based UNIX processor for real-time control of Contel's local area network, a product called ContelNet (tm).

The following presentation describes the major stumbling blocks encountered as well as the solutions developed to overcome these stumbling blocks.

SYSTEM ACQUISITION

Number of vendors found with finished real-time UNIX adaptations: 0
Number of vendors found with finished standard UNIX versions: 1

We acquired a Multibus-based microcomputer with a M68000 CPU chip, lots of RAM, a smallish Winchester disk, a floppy disk drive and a cartridge tape backup. The system runs UNIX Version 7 with the Berkeley enhancements. We have a binary license only, although we have acquired the capability to add (and reconfigure) device driver routines.

The latter capability includes a set of existing device drivers, certain necessary INCLUDE files, a make file facility, all surrounding a central UNIX.0 file. By adding our own device driver routines, we can then link and generate a new /UNIX file.

OUR NETWORK CONTROLLER

Our network controller is to consist of a resident portion with the capability of receiving real-time stimuli from the network, responding to these stimuli, monitoring and displaying network status on a screen, plus upwards of 60 operator commands to perform configuration, control, monitoring, display, print, and other service functions.

The interface to the network is to be through CONTEL's own Multibus* interface processor board, a Z80-based CPU with its own RAM and EPROM, its own DMA, as well as serial and parallel ports. Connected to this CPU board is either an RF modem or a baseband transceiver board. Contel offers the choice of baseband or broadband systems, including Ethernet systems.

EARLY DECISIONS

For maximum speed, we chose to feed information to and from the UNIX processor directly via the Multibus. This is much faster than routing information through serial I/O ports. The two CPUs on the Multibus are arranged in a multi-master configuration, with parallel priority resolution of Multibus accesses.

The order of priority of accesses to the Multibus is:

(1) Winchester Disk Controller,
(2) Floppy Disk Controller,
(3) ContelNet Z-80 CPU,
(4) M68000 UNIX CPU,
(5) Cartridge Tape Controller,
(6) Octal Serial I/O Board.

A 128-K RAM board is designated to hold all common interface information for inter-CPU communication.

Problem : How do we manage this 128-K RAM under UNIX?

Solution: Hide it from UNIX sizing algorithm by means of a gap in addressing, then map its physical address space into virtual process address space by means of the PHYS function call.

* Registered trademark of Intel Corp.

The result is a 128-K RAM area not under UNIX memory management, and therefore freely available for implementation of inter-CPU protocols.

INTER-CPU COMMUNICATION

For communication between the UNIX CPU and the Z-80 CPU we establish two unidirectional ring buffers. Each CPU has its own buffer pointers. All messages and orders stored in the buffers are handled on a FIFO basis.

Each message in buffer has a length byte followed by text. All free buffer bytes are maintained as zeros. Zero length means (implicitly) that there is no message there. All bytes are cleared to zeros when taken from the buffer. Length bytes are always written last and also cleared last, in order to prevent the reader process from taking incomplete messages.

A status control byte is also established, with the following states:

(0) Waiting for Z-80 to carry out a software RESET,

(1) Z-80 is reset, now waiting for its program to be loaded in the buffer,

(2) UNIX CPU has loaded Z-80's program in buffer,

(3) Z-80 has taken its own operational program and relocated it in its ultimate destination. Buffer is cleared and pointers readied for operation,

(4) The other buffer has also been cleard and pointers readied for operation.

Normal status is (4). Any anomaly causes the control byte to recycle to (0). Within the network controller, steps 0 through 3 are performed by EPROM-based software.

UNIX INTER-PROCESS COMMUNICATION

The resident UNIX process grew quickly to a size that required partitioning. At present, there are five resident background UNIX processes:

1. Network Input Process,
2. Network Output Process,
3. UNIX File Server Process,
4. Display Server Process, and
5. Print Server Process.

Problem : How to communicate between these processes in an efficient and safe manner?

Signals and pipelines could not be used. Signals are too limited in scope, and pipelines are (1) too slow, and (2) unsafe, in that reader and writer can both go to sleep at the most inopportune times.

Solution : Use portions of the hidden memory to set up inter-process communication buffers among pairs of UNIX processes, just as was the case for communication between UNIX and the Z-80.

A COMMON DATA BASE IN RAM

In addition to fast inter-process communication, we also require that certain data tables be accessible to all resident processes in true random-access fashion. This can be accomplished by mapping the common data base elements into additional portions of the hidden RAM. An address base is declared and all offsets are calculated from the known lengths of each element. For example,

```
    long *Table = (long *) (VIRTBASE + TABLEOFFSET);
```

This statement defines a pointer to a table of long integers. Each process carries out the same PHYS function call, and then defines the same data base elements.

ACCESS CONTROL

Problem : We need a central protection device to control access to certain data base elements while they are being modified (updated). We shall allow multiple data base readers, but only one writer at a time.

Solution: The solution to this problem evolved in stages. In the first stage we used locking and unlocking semaphores, the control of which was placed into the UNIX kernel so as to guarantee freedom from interruption by sibling processes. This worked but seemed clumsy.

Later, the search, insert and delete functions were implemented in the UNIX kernel itself. This improved efficiency and made the code more readable. Finally, in stage three, we placed the access control tables themselves into the UNIX kernel, including two binary search tables and a direct access table. The final result is a fast and streamlined data base access mechanism.

PROCESS PRIORITY CONTROL

Problem : We needed to control process priorities among the five resident processes and other transient processes.

Experiments with NICE proved inconclusive. Some limited amount of control is possible, but seemed unstable when viewed over long periods of time, i.e., when applied to a resident process. UNIX was written as an egalitarian time-sharing operating system and does not offer positive priority control devices.

Solution : We exercise a certain amount of control programmatically, and leave the use of NICE as a last resort. To date, we have not found it to be necessary.

Each resident process is structured as an infinite loop which continuously times itself to assess its own rate of progress. Then, certain operations of secondary importance are skipped if progress is slow, i.e., the CPU is very busy. Furthermore, each process relinquishes the CPU if it does not have too much to do.

This technique affords but limited control over process priorities, and it is good only if the time constants of process activities and responses are of the same order of magnitude as those of the UNIX time slicer, i.e., one second. On a millisecond scale this technique is not useful.

We are working on a refinement of this technique which would allow for more accurate control of task execution. So far, we cannot report any significant progress made.

LOCKING PROCESSES IN RAM

Problem : UNIX must not be allowed to swap resident processes. This would be much too slow for real-time network control. How can we avoid swapping?

Solution : First, we provide enough RAM to make sure that swapping is not necessary. Second, we LOCK processes in RAM.

RUNNING FAST

Problem : How to minimize disk accesses and speed up operation of the system, given UNIX's file structure?

Solution : First, we keep all operational files OPEN for the resident file server. Second, we eliminate unnecessary internal buffering by means of SETBUF (NULL), and third, all disk I/O operations use 512-byte buffers.

PERMISSIONS AND SUCH

Many of the function calls, such as the PHYS call, are privileged calls for ROOT only. At first, this seems to imply that all resident processes should belong to ROOT. The entire network control system would then run as ROOT. This seemed unfortunate, because it means that the implicit protection afforded by UNIX in its privileged vs. non-privileged functions would be lost.

To circumvent this problem, we implemented a means of changing the effective user id of each process during processing. At its inception, each process begins as ROOT and performs the privileged functions required for network operations. Then, a

SETUID call is made, reverting the user id to an ordinary user. In order to make this possible, each load module must be made to belong to ROOT and its mode must be changed by means of the CHMOD function to be 4755. Both the CHOWN and the CHMOD calls are made from within the MAKEFILE at the time when the subject module is linked.

Additional problems may arise with permissions dealing with data files. All files to be written are opened while still in privileged mode. This does not affect the ability to write later, following the SETUID.

DEBUGGING

In order to debug the complex assemblage of processes, a debugging byte was established in the RAM data base for each resident application process. Four bits of each debugging byte were defined, as follows:

- Echo major input to process,
- Echo minor input to process,
- Echo major output from process,
- Echo minor output from process.

Each process sends debugging information to the console terminal if the appropriate bit of its debugging byte is set. A command allows debugging flags to be set easily from the console terminal.

SUMMARY

In summary, the original goals of the network control system have been attained:

(1) Two CPUs are communicating directly through RAM over the Multibus,

(2) Fast inter-process communication is achieved among resident UNIX processes,

(3) A common RAM-resident data base is directly accessible to all UNIX processes,

(4) UNIX processes dynamically adjust their own time slices as needed, and

(5) Disk I/O is minimized and resident processes made to be stationary in RAM.

Real-Time Extensions to the UNIX Operating System

Byron Look, Gary Ho
Hewlett-Packard
Data Systems Division
11000 Wolfe Road
Cupertino, CA 95014

1. Introduction

This paper discusses real time extensions to the UNIX[1] operating system. The real-time requirements of technical computer systems are first discussed. The deficiencies of the UNIX operating system for real-time applications are then described. Finally, proposals for providing real-time extensions to UNIX are presented.

2. Real-Time Requirements for Technical Computers

A real-time system is a system which can respond predictably and in a timely manner to events in the real world. Specific response time requirements are application dependent; the more stringent the time constraints, the more real-time in nature the application is. Typical real-time applications include process control, high speed data acquisition, machine monitoring, instrumentation and lab automation. The remainder of this section details specific real-time capabilities required in these application areas.

2.1. Scheduling of Real-Time Processes

Priority-based scheduling of real-time processes is critical in situations where it is more important to execute a pending process than to continue executing the current process. A typical example of this type of situation is the occurrence of an alarm condition requiring immediate action. The operating system must be able to recognize the condition (usually through an interrupt), preempt the currently executing process and perform a fast context switch to allow the higher priority process to execute. A real-time operating system is typically able to schedule a process on the basis of events occurring, including a device interrupt, a software interrupt, at prescribed times or after prescribed offsets in time. User control of priorities (setting and resetting) needs to be allowed in a real-time operating system. When no real-time processes are pending, it should be possible to schedule the CPU in a time-sliced manner. It should also be possible to establish software priorities which allow real-time processes to run at a higher priority than some devices.

2.2. Guaranteed Interrupt Response

Providing optimal real-time response requires that the computer system quickly recognize the occurrence of an event and take action based on this event within the required time. A real-time operating system must be able to respond to both hardware and software interrupts. The operating system itself should be interruptible and reentrant. There should be minimal overhead in the operating system; the operating system should

[1] UNIX is a trademark of Bell Laboratories.

not get in the way of a high-priority, real-time process. The longest path through privileged portions of the operating system therefore must be minimized. In particular, it is crucial to minimize the time that the interrupt system is turned off and to insure that high priority device drivers are not impeded from quick servicing of interrupts.

2.3. High Speed Data Acquisition

In high speed data acquisition applications, the computer system must be able to handle sustained data burst rates up to and exceeding 20 Mbytes/second. To satisfy these performance requirements, real-time operating systems need to minimize the amount of extraneous processing such as intermittent allocation of disc space, excess movement of the disc heads, moving data from system buffers to user buffers and limits on the size of buffers. Hence, features required include the ability to preallocate contiguous disc space, provisions for user control over buffering, and allowing for direct transfer of data to or from user buffers.

2.4. User Control of System Resources

A key characteristic of real-time systems is the ability to provide users with specific control of the system resources including the CPU, memory and I/O. Control of the CPU is discussed above under scheduling and includes direct user control of process priorities, preemptive scheduling and time-based scheduling. In addition, the user should be able to lock a process in memory for faster context switches when an interrupt occurs. The user should also be able to control and guarantee memory allocation for buffers. Locking and unlocking devices and files is also of importance.

2.5. I/O

Real-time applications typically involve interfacing to a multitude of I/O interfaces. Provisions must be made for developing and easily incorporating custom or user-written I/O drivers into the system. For standard devices, it should be possible to implement standard I/O device libraries. Direct software control of I/O must also be provided. Processes should be allowed to perform I/O asynchronously to devices including I/O to preallocated disc space. To provide precise control over the system, it is also important to allow for non-blocking I/O. Prioritized servicing of I/O requests should also be possible.

2.6. Inter-Process Communications

Real-time applications often require an efficient, general inter-process communications facility. Messages should be sendable between arbitrary processes. Process control applications, for example, are often typified by multiple, dedicated processes which communicate asynchronously with one another. Synchronization mechanisms (semaphores, monitors) are thus necessary. Frequently, shared memory is desirable for these tightly coupled asynchronous processes as an efficient means for sharing data.

2.7. General Features Important in Real-Time Applications

o High reliability and high system availability (power fail recovery, fault tolerance, error detection)

o Data communications and good interconnectivity with existing machines

o Data base management

o Interactive graphics for data analysis and presentation

o Performance support via microcode, specialized hardware

3. UNIX in Real-Time Applications

This section discusses the suitability of UNIX for real-time applications, based on the market requirements presented in the previous section.

3.1. Process Management

The UNIX scheduling algorithm is designed to provide equitable access to the CPU and memory. It was designed for timesharing environments, hence processes are time sliced. Scheduling priorities of processes are recalculated periodically to ensure a fair share of the CPU by each process.

UNIX could be modified to allow users to have direct control over process priorities. For instance, the priority of "real-time" processes could be made to be fixed, not varied by the UNIX operating system. Real-time processes of equal priority should be executed in a round robin fashion and not be subject to time slicing.

3.2. Interrupt Handling

Interrupt scheduling is done via signals (software interrupts) from the driver, but the reliability of signals is jeopardized by the existence of race conditions. The signaling mechanisms of Berkeley UNIX have resolved most of the problems.

A UNIX process that is executing a system call cannot be preempted and much of the UNIX kernel code depends on this assumption. For faster context switching, some of the longer UNIX system calls (such as execve or fork which copy large amounts of data) could be made to be interruptible and reentrant (for example, after each block of data is transferred). This improves the responsiveness of the system in servicing higher priority requests.

3.3. File System

The UNIX operating system allocates disc file space for efficient disc utilization. Space is not allocated upon file creation but instead is allocated as needed on write operations. This implies not being able to guarantee the disc space will actually be available when it is needed. It also implies that performance on writes will be slower since time may have to be spent to allocate the space during the write operation. Another consequence is the probability that the disc file will not be contiguous, possibly also resulting in more disc seeking and lower performance. The performance of process creation can be improved significantly if the object file is stored contiguously on disc. This is especially true in Bell System UNIX as the whole process must be in memory before it can start execution. Sequentiality of disc writes is also an issue, as there is no guarantee by UNIX that writes will actually be posted to the disc in the same order they were requested.

The disc sort routine in UNIX could be modified to give preferential treatment to real-time disc I/O requests over other I/O. The raw disc capability of UNIX could potentially be used for direct disc I/O. Extend files and contiguous files should be supported to speed up disc accesses.

3.4. User Control of System Resources

System III UNIX does not provide a means of locking a process in memory. This feature has been added to System V. Locking a process in memory is desirable for fast process dispatching in response to real-time events. Additionally, UNIX does not provide a direct means of locking a device.

U.C. Berkeley's 4.2BSD UNIX offers the flock primitive which provides processes with the capability of locking a file on an advisory basis. The /user/group UNIX standard offers lockf primitive which supports file locking on the byte level. This incompatibility should be resolved.

3.5. I/O

UNIX does not provide for asynchronous I/O operations, nor is there a capability for connecting directly to I/O devices. System V does, though, provide for non-blocking reads and for non-blocking writes. Write-through writes should be supported to force posting of data to disc as soon as possible. Although a write followed immediately by a fsync system call has the same effect, it involves two system calls. Real-time applications may involve the interfacing of custom I/O devices and drivers. This raises the issue of provisions for skeleton drivers and the UNIX source license ramifications of this.

U.C. Berkeley's 4.2BSD provides calls for asynchronous, non-blocking I/O operations.

3.6. Inter-Process Communications

System III UNIX does not support a generalized interprocess communication (IPC) package. (System V supports both shared memory and messages.) Pipes and signals can be used as communications mechanisms. A process can send a signal to another process, thereby interrupting the signaled process. The signal handler can either ignore or catch the signal. There are some limitations: there are a limited number of signals; signals are not buffered and perhaps most important, signals can be unreliable. Specifically, it is possible for a second signal to arrive before a prior signal is reset, causing the second signal not to be caught.

The Berkeley version of UNIX fixes the reliability problems of signals, though there is still a limit to the number of signals possible (32). 4.2BSD UNIX also provides for some shared memory support. It also provides a set of interprocess communication mechanisms which is quite general and can be used both for inter and intra processor communication.

3.7. Real-Time Privileges

Real-time privileges are the privileges given to processes to become real-time processes and lock themselves in memory. These privileges must be controlled carefully. A possible approach to control real-time privileges is to associate real-time privileges to (Berkeley) access groups. These real-time access groups could be specified by the super user. A process will be given real-time privileges if it is a member of the real-time access groups. This approach allows dynamic revocation of real-time privileges by redefining the real-time access groups and provides tight control on the real-time privileges.

4. Summary

UNIX is designed for time shared applications. In its current form, it has many shortcomings for real-time applications. Many of the real-time requirements discussed in this paper are satisfied by either System V or Berkeley 4.2 UNIX. However, neither of them can be considered as a good real-time operating system. A real-time UNIX operating system can be constructed by combining the functionalities offered by the two and adding real-time capabilities to the system.

5. References

1. A. Boxer, L. Gale and T. Teixeira, "Reliable Data Acquisition with a UNIX Based Laboratory Computer", UNIX Based Systems for Process and Laboratory Control, WESCON, September 1982.

2. H. Cohen and J.C. Kaufeld, "The Network Operations Center System", The Bell System Technical Journal, Vol. 57, No. 6, July-August 1978, 2289-2304.

3. D.M. Harland, "High Speed Data Acquisition: Running a Realtime Process and a Time-shared System (UNIX) Concurrently", Software - Practice and Experience, Vol. 10, 1980, 273-281.

4. A.V. Hays, B.J. Richmond and L.M. Optican, "A UNIX-Based Multiple-Process System for Real-Time Data Acquisition and Control", UNIX Based Systems for Process and Laboratory

5. W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, "4.2BSD System Manual", Draft of February 14, 1982, Dept. of EECS, University of California, Berkeley.

6. H. Lycklama and D.L. Bayer, "The MERT Operating System", The Bell System Technical Journal, Vol. 57, No. 6, July-August 1978, 2049-2086.

7. H. Lycklama, "UNIX on a Microprocessor", The Bell System Technical Journal, Vol. 57, No. 6, July-August 1978, 2087-2101.

8. L.G. McKnight, "CONCEPS-II - Process Monitor for UNIX Operated Computers", UNIX Based Systems for Process and Laboratory Control, WESCON, September 1982.

9. A. Romberger, "Methods for Real-Time Speech Processing on UNIX", UNIX Based Systems for Process and Laboratory Control, WESCON, September 1982.

10. L. Yencharis, "Menu driven real-time system relieves data acquisition headaches", Electronic Products, August 18, 1982.

Author Index

Thomas Alborough
Reloadable UNIX Device Drivers 135

Helene Armitage et al
Proposed Syntax Standard for UNIX System Commands .. 89

Jeffrey S. Barr et al
Life with UNIX in Real-Time 285

Bruce Borden et al
The Excelan TCP/IP Protocol Package 1

Ross Bott
OSx: Towards a Single UNIX System for Superminis 145

Chet Britten et al
Multiprocessor Debugging on a Shared Memory System .. 33

Richard N. Bryant et al
Writing Device Drivers for XENIX Systems 195

Jerry Carlin et al
Building Tunnels and Bridges:
Constructing a Commercial Application Under UNIX 259

Paul Chen et al
Multiprocessor Debugging on a Shared Memory System .. 33

Curtis B. Downing et al
Transparent Implementation of Shared Libraries 209

Frank Farance et al
Transparent Implementation of Shared Libraries 209

Alan R. Feuer
You CAN Feel Good Knowing It Is Written in C 57

Matthew S. Hecht et al
A Distributed File System for UNIX 23

Kathleen Hemenway et al
Proposed Syntax Standard for UNIX System Commands ... 89

Gary Ho et al
Real-Time Extensions to the UNIX Operating System 293

P. Honeyman et al
Experimental Implementation of UUCP —
Security Aspects 245

Robert J.K. Jacob
User-Level Window Managers for UNIX 123

Rob Kolstad et al
Mapping the UUCP Network 251

Bob Kridle
New 1/2-Inch Tape Options and Trade-Offs for
4.2BSD UNIX on DEC VAX Processors 168

John R. Levine et al
A Distributed File System for UNIX 23

Jeff Lindberg
A Layered Implementation of the UNIX Kernel
on the HP9000 Series 500 Computer 183

Daniel B. Long et al
CSNET Grows up 13

Byron Look et al
Real-Time Extensions to the UNIX Operating System 293

Heinz Lycklama
/usr/group Standards Effort 231

Rex McDowell
A UNIX-Based Color Graphics Workstation 115

Jean McNamara et al
Writing Device Drivers for XENIX Systems 195

Lambert Meertens et al
An Implementation of the B Programming Language 65

Joaquin Miller
Connecting a UNIX System to an X.25 Network 47

Bill Northlich et al
The Excelan TCP/IP Protocol Package 1

D.A. Nowitz et al
Experimental Implementation of UUCP —
Security Aspects 245

Michael T. O'Brien et al
CSNET Grows Up 13

Pat E. Parseghian et al
Behind Every Binary License is the UNIX Heritage 75

Steven Pemberton et al
An Implementation of the B Programming Language 65

Steven T. Polyak et al
Life with UNIX in Real-Time 285

T. Scott Pyne et al
MPS: A UNIX-Based Microcomputer Message Switching System 99

Brian E. Redman et al
Behind Every Binary License is the UNIX Heritage 75

B.E. Redman et al
Experimental Implementation of UUCP — Security Aspects 245

David Robboy
UNIX Block I/O Optimization on Microcomputers 223

Charles M. Robins
Adapting UNIX for Data Communications 39

Joseph L. Steffen
Software Administration over Computer Networks — The Exptools Experience 17

Karen Summers-Horton et al
Mapping the UUCP Network 251

Andrew Tannenbaum
Political History of UNIX 83

Page Thompson et al
Building Tunnels and Bridges: Constructing a Commercial Application Under UNIX 259

Ellen Ullman et al
Building Tunnels and Bridges: Constructing a Commercial Application Under UNIX 259

Paresh Vaish et al
Writing Device Drivers for XENIX Systems 195

Justin C. Walker et al
A Distributed File System for UNIX 23

Joseph S.D. Yao et al
MPS: A UNIX-Based Microcomputer Message Switching System 99

Keyword Index to Titles

Analysis
Behind Every Binary License is the UNIX Heritage 75
Political History of UNIX 83

Applications
Building Tunnels and Bridges:
Constructing a Commercial Application Under UNIX 259
MPS: A UNIX-Based Microcomputer Message
Switching System 99

Benchmarks
New 1/2-Inch Tape Options and Trade-Offs for
4.2BSD UNIX on DEC VAX Processors 168

B Language
An Implementation of the B Programming Language 65

C Language
You CAN Feel Good Knowing It Is Written in C 57

Communications
Adapting UNIX for Data Communications 39
MPS: A UNIX-Based Microcomputer Message
Switching System 99

Distributed File Systems
A Distributed File System for UNIX 23

Drivers
Reloadable UNIX Device Drivers 135
Writing Device Drivers for XENIX Systems 195

Graphics
A UNIX-Based Color Graphics Workstation 115

HP-UX
A Layered Implementation of the UNIX Kernel
on the HP9000 Series 500 Computer 183

I/O Optimization
UNIX Block I/O Optimization on Microcomputers 223

Multiprocessor Systems
Multiprocessor Debugging on a Shared Memory System .. 33

Networking
Connecting a UNIX System to an X.25 Network 47
CSNET Grows Up 13
A Distributed File System for UNIX 23
The Excelan TCP/IP Protocol Package 1
Experimental Implementation of UUCP —
Security Aspects 245
Mapping the UUCP Network 251
Software Administration over Computer Networks —
The Exptools Experience 17

Protocols
The Excelan TCP/IP Protocol Package 1

Real-Time
Life with UNIX in Real-Time 285
Real-Time Extensions to the UNIX Operating System 293

Security
Experimental Implementation of UUCP —
Security Aspects 245

Shared Libraries
Transparent Implementation of Shared Libraries 209

Standards
Proposed Syntax Standard for UNIX System Commands .. 89
/usr/group Standards Effort 231

System V UNIX
OSx: Towards a Single UNIX System for Superminis 145

TCP/IP
The Excelan TCP/IP Protocol Package 1

UUCP
Experimental Implementation of UUCP —
Security Aspects 245
Mapping the UUCP Network 251

Windowing
User-Level Window Managers for UNIX 123

Workstations
A UNIX-Based Color Graphics Workstation 115

XENIX
Writing Device Drivers for XENIX Systems 195

X.25 Networks
Connecting a UNIX System to an X.25 Network 47
CSNET Grows up 13

4.2BSD
New 1/2-Inch Tape Options and Trade-Offs for 4.2BSD UNIX on DEC VAX Processors 168
OSx: Towards a Single UNIX System for Superminis 145

The USENIX Association

USENIX, the UNIX and Advanced Computing Systems professional and technical organization, is a not-for-profit membership association made up of systems researchers and developers, systems administrators, programmers, support staff, application developers and educators.

USENIX is dedicated to:

* fostering innovation and communicating research and technological developments,
* sharing ideas and experience, relevant to UNIX, UNIX-related and advanced computing systems
* providing a forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, the Association sponsors two annual technical conferences and frequent symposia and workshops addressing special interest topics, such as C++, distributed systems, Mach, systems administration, and security. USENIX publishes proceedings of its meetings, a bi-monthly newsletter *login:*, and a refereed technical quarterly, *Computing Systems*. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts.

There are four classes of membership in the Association: student, individual, institutional, and supporting, differentiated primarily by the fees paid and services provided. The supporting members of the Association are:

Digital Equipment Corporation
Frame Technology, Inc.
Matsushita Graphic Communication Systems, Inc.
mt Xinu
Open Software Foundation
Quality Micro Systems
Rational Corporation
Sun Microsystems, Inc.
Sybase, Inc.
UNIX System Laboratories, Inc.
UUNET Technologies, Inc.

For further information about membership or to order publications, contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710-2565
Telephone: 510/528-8649
Email: office@usenix.org
Fax: 510/548-5738

4655 Old Ironsides Drive
Suite 200
Santa Clara, CA 95050

UNIX